RealView® 编译工具

3.1 版

汇编程序指南

ARM DUI 00204HC

RealView 编译工具

汇编程序指南

版本信息

本手册进行了以下更改。

更改历史记录

日期	发行号	保密性	更改
2002 年 8 月	A	非保密	1.2 版
2003 年 1 月	B	非保密	2.0 版
2003 年 9 月	C	非保密	RVDS v2.0 的 2.0.1 版
2004 年 1 月	D	非保密	RVDS v2.1 的 2.1 版
2004 年 12 月	E	非保密	RVDS v2.2 的 2.2 版
2005 年 5 月	F	非保密	RVDS v2.2 SP1 的 2.2 版
2006 年 3 月	G	非保密	RVDS v3.0 的 3.0 版
2007 年 3 月	H	非保密	RVDS v3.1 的 3.1 版。

所有权声明

保密状态

本文档的内容是非保密的。根据 ARM 与 ARM 将本文档交予的参与方的协议条款，使用、复制和公开本文档内容的权利可能会受到许可限制的制约。

产品状态

本文档的信息是开发的产品的最新信息。

网址

`http://www.arm.com`

目录
RealView 编译工具
汇编程序指南

前言

本前言介绍 *《RealView 编译工具汇编程序指南》*。它包含以下几节:

- 第 viii 页的*关于本手册*
- 第 xi 页的*反馈*

关于本手册

本手册提供有关 *RealView® 编译工具* (RVCT) 汇编程序的指导和参考信息。这包括 `armasm`、独立的汇编程序以及 C 和 C++ 编译器中的嵌入式汇编程序。本手册介绍汇编程序的命令行选项，可供汇编语言程序员使用的汇编语言助记符、伪指令、宏和指令。

适用对象

本手册是为所有使用 RVCT 编写应用程序的开发者编写的。本手册假定您是一位有经验的软件开发人员，并且熟悉 RealView 编译工具要点指南中所述的 ARM 开发工具。

使用本手册

本手册由以下章节组成：

第 1 章 *简介*

本章简要介绍了 RVCT 汇编程序和汇编语言。

第 2 章 ***编写 ARM 汇编语言***

本章提供了可帮助您使用 ARM 汇编程序和汇编语言的指导信息。

第 3 章 *汇编程序参考*

本章介绍了有关 ARM 汇编程序提供的语法和结构的参考材料。

第 4 章 ***ARM 和 Thumb 指令***

本章介绍了有关 ARM 和 Thumb 指令集的参考材料，涵盖了 Thumb-2 和以前版本的 Thumb 以及 Thumb-2EE。

第 5 章 ***NEON 和 VFP 编程***

本章介绍了有关 ARM NEON™ 技术和 VFP 指令集的参考材料。本章还介绍了其他 VFP 特定的汇编语言信息。

第 6 章 ***无线 MMX 技术指令***

本章介绍了有关 ARM 对无线 MMX™ 技术的支持的参考材料。

第 7 章 *指令参考*

本章介绍了有关可在 ARM 汇编程序 `armasm` 中使用的汇编程序指令的参考材料。

本手册假定 ARM 软件安装在缺省位置，例如，在 Windows 中的路径为 `volume:\Program Files\ARM`。引用路径名时，假定安装位置为 *install_directory*，例如 `install_directory\Documentation\...`。如果将 ARM 软件安装在其他位置，则可能需要更改此位置。

印刷约定

本手册使用了以下印刷约定：

`monospace` 表示可以从键盘输入的文本，如命令、文件和程序名以及源代码。

`monospace`（下划线部分为 mono） 表示允许的命令或选项缩写。可只输入下划线标记的文本，无需输入命令或选项的全名。

`monospace italic`

表示此处的命令和函数的自变量可用特定值代替。

等宽粗体

表示在示例代码以外使用的语言关键字。

斜体 突出显示重要注释、介绍特殊术语以及表示内部交叉引用和引文。

粗体 突出显示界面元素，如菜单名称。有时候也用在描述性列表中以示强调，以及表示 ARM 处理器信号名称。

更多参考出版物

本部分列出了 ARM 公司和第三方发布的、可提供有关 ARM 系列处理器开发代码的附加信息的出版物。

ARM 将定期对其文档进行更新和更正。有关最新勘误表、附录和 ARM 常见问题解答，请访问 `http://www.arm.com`。

ARM 公司出版物

本手册包含的参考信息专用于随 RVCT 提供的开发工具。该套件中包含的其他出版物有：

- *《RVCT 要点指南》*(ARM DUI 0202)
- *《RVCT 编译器用户指南》*(ARM DUI 0205)
- *《RVCT 编译器参考指南》*(ARM DUI 0348)
- *《RVCT 库和浮点支持指南》*(ARM DUI 0349)
- *《RVCT 链接器和实用程序指南》*(ARM DUI 0206)

- *《RVCT 开发指南》* (ARM DUI 0203)
- *《NEON 向量化编译器指南》* (ARMDUI 0350)
- *《RealView Development Suite 词汇表》* (ARM DUI 0324)。

有关基本标准、软件接口以及 ARM 支持的标准的完整信息，请参阅 `install_directory\Documentation\Specifications\...`。

此外，有关与 ARM 产品相关的特定信息，请参阅下列文档：

- *《ARM6-M 体系结构参考手册》* (ARM DDI 0419)
- *《ARM7-M 体系结构参考手册》* (ARM DDI 0403)
- *《ARM 体系结构参考手册》，ARMv7-A 和 ARMv7-R 版* (ARM DDI 0406)
- *《ARM 体系结构参考手册 Thumb®-2 补充》* (ARM DDI 0308)
- *《ARM 体系结构参考手册安全扩展补充》* (ARM DDI 0309)
- *《ARM 体系结构参考手册 Thumb-2 执行环境补充》* (ARM DDI 0397)
- *《ARM 体系结构参考手册高级 SIMD 扩展和 VFPv3 补充》* (ARM DDI 0268)
- *《ARM 参考外围设备规范》* (ARM DDI 0062)
- 您的硬件设备的 ARM 数据表或技术参考手册。

其他出版物

有关 ARM 体系结构的介绍，请参阅 Steve Furber 编著的 *《ARM 片上系统体系结构》*（第 2 版，2000）。Addison Wesley，ISBN 0-201-67519-6。

有关 Intel® 无线 MMX™ 技术的完整信息，请参阅 *《无线 MMX 技术开发指南》*（2000 年 8 月），订单号：251793-001，可从 `http://www.intel.com` 获取。

反馈

ARM 公司欢迎用户就 RealView 编译工具及其文档提供反馈意见。

RealView 编译工具的反馈信息

如果您有关于 RVCT 的任何问题，请与您的供应商联系。为便于他们快速提供有用的答复，请提供：

- 您的姓名和公司
- 产品序列号
- 您所用版本的详细信息
- 您运行的平台的详细信息，如硬件平台、操作系统类型和版本
- 能重现问题的一小段独立代码示例
- 您预期发生和实际发生的情况的详细说明
- 您使用的命令，包括所有命令行选项
- 能说明问题的示例输出
- 工具的版本字符串，包括版本号和内部版本号

关于本手册的反馈

如果您发现本手册有任何错误或遗漏之处，请发送电子邮件到 `errata@arm.com`，并提供：

- 文档标题
- 文档编号
- 您有疑问的页码
- 问题的简要说明

我们还欢迎您对需要增加和改进之处提出建议。

第 1 章
简介

本章介绍随 *RealView®* *编译工具* (RVCT) 提供的汇编程序。它包含以下一节：

1.1 关于 RealView 编译工具汇编程序

RVCT 具有:

- 独立的汇编程序 `armasm`，本指南将予以介绍。
- 内置到 C 和 C++ 编译器中的优化内联汇编程序和非优化嵌入式汇编程序。这些汇编程序使用相同的汇编指令语法，但是不在本指南中进行介绍。有关内联和嵌入式汇编程序的详细信息，请参阅 *RealView 编译工具开发指南*中的*混合使用 C、C++ 和汇编语言*一章。

如果要将 RVCT 的旧版本升级，请阅读 RealView 编译工具要点指南，以获得此版本的新功能以及改进提高的相关信息。

1.1.1 ARM 汇编语言

当前的 ARM/Thumb 汇编语言已经取代了较早版本的 ARM 和 Thumb 汇编语言。

使用当前语言编写的代码可针对 ARM、Thumb 或 Thumb-2 进行汇编。如果使用了不可用的指令，汇编程序会报错。

1.1.2 无线 MMX 技术指令

汇编程序支持 Intel® 无线 MMX™ 技术指令，可汇编代码以运行于 PXA270 处理器上。此处理器实现具有 MMX 扩展的 ARMv5TE 体系结构。RVCT 支持无线 MMX 技术控制和*单指令多数据* (SIMD) 数据寄存器，并且包含了用于无线 MMX 技术开发的新指令。此外，它还改善了对加载和存储指令的支持。有关 RVCT 中无线 MMX 技术支持的信息，请参阅第 6 章 *无线 MMX 技术指令*。

1.1.3 NEON 技术

ARM NEON™ 技术是 ARMv7 体系结构的可选组件。NEON 是针对高级媒体和信号处理应用程序以及嵌入式处理器的 64/128 位混合 SIMD 技术。它是作为 ARM 内核的一部分实现的，但有自己的执行管道和寄存器组，该寄存器组不同于 ARM 核心寄存器组。

NEON 支持整数、定点和单精度浮点 SIMD 运算。这些指令在 ARM 和 Thumb-2 中都可用。

有关 NEON 的详细信息，请参阅第 5 章 *NEON 和 VFP 编程*。

1.1.4 使用示例

本手册引用了 RealView Development Suite 随附的示例，这些示例位于主示例目录 `install_directory\RVDS\Examples` 中。有关所提供示例的汇总，请参阅*《RealView Development Suite 入门指南》*。

第 2 章
编写 ARM 汇编语言

本章介绍了编写 ARM® *汇编语言* 的一般原则，其中包含以下几节:

2.1 简介

本章提供了如何编写 ARM 汇编语言模块的实用基础知识。同时还提供了与 ARM 汇编程序 (`armasm`) 所提供的工具有关的信息。

本章未提供 ARM、Thumb®-2、Thumb、NEON™、VFP 或 MMX 指令集的详细描述。有关这些信息，请参阅：

- 第 4 章 *ARM 和 Thumb 指令*
- 第 5 章 *NEON 和 VFP 编程*
- 第 6 章 *无线 MMX 技术指令*。

有关详细信息，请参阅《*ARM 体系结构参考手册*》。

为了给熟悉 RVCT2.1 及更早版本中支持的 ARM 和 Thumb 汇编语言的编程人员提供方便，本章专门在一节中概括了这些版本与 ARM 汇编语言的最新版本之间的区别。请参阅第 2-51 页的*汇编语言变更*。

2.1.1 代码示例

本章包含许多代码示例，其中许多示例位于 `*install_directory*\RVDS\Examples\...\asm` 目录中。

若要生成并链接汇编语言文件，请按照下列步骤操作：

1. 在指令提示符处键入 `armasm --debug *filename*.s`，以汇编该文件并生成调试表。
2. 键入 `armlink *filename*.o -o *filename*` 以链接该目标文件并生成一个 ELF 可执行映像。

若要执行和调试该映像，请使用适当的调试目标（如 *RealView 指令集模拟器* (RVISS)），将该映像加载到兼容的调试器（如 RealView Debugger）中。

若要查看汇编程序如何转换源代码，请输入：

```
fromelf -c filename.o
```

有关 `armlink` 和 `fromelf` 的详细信息，请参阅《*RealView 编译工具链接器和实用程序指南*》。

2.2 ARM 体系结构概述

本节简要概述了 ARM 体系结构。

ARM 处理器是典型的 RISC 处理器，因为它们执行的是加载/存储体系结构。只有加载和存储指令才能访问内存。数据处理指令只操作寄存器的内容。

本节介绍了以下内容:

- *体系机构的版本*
- *ARM、Thumb、Thumb-2 和 Thumb-2EE 指令集*
- 第 2-4 页的*ARM、Thumb 和 ThumbEE 状态*
- 第 2-5 页的*处理器模式*
- 第 2-6 页的*寄存器*
- 第 2-8 页的*指令集概述*
- 第 2-9 页的*指令功能*。

2.2.1 体系机构的版本

本手册中的信息和示例假定您使用的是执行 ARMv4 或更高版本体系结构的处理器。所有这些处理器都具有 32 位寻址范围。

有关各种体系结构版本的详细信息，请参阅 *《ARM 体系结构参考手册》*。

2.2.2 ARM、Thumb、Thumb-2 和 Thumb-2EE 指令集

ARM 指令集是一组提供一整套运算的 32 位指令。

ARMv4T 及更高版本定义了一个名为 Thumb 指令集的 16 位指令集。32 位 ARM 指令的多数功能都可用，但有些运算需要与其他指令结合使用。Thumb 指令集提供了更好的代码密度，但会损害性能。

ARMv6T2 定义了 Thumb-2，它与 Thumb 指令集相比有了重大改进。Thumb-2 提供了几乎与 ARM 指令集完全相同的功能。它同时具有 16 位和 32 位指令，并可同时实现类似于 ARM 的性能以及类似于 Thumb 的代码密度。

在 ARMv6 及更高版本中，所有 ARM 和 Thumb 指令都是小端的。在 ARMv6T2 及更高版本中，所有 Thumb-2 指令获取也都是小端的。

ARMv7 定义了 Thumb-2 执行环境 (Thumb-2EE)。Thumb-2EE 指令集基于的是 Thumb-2，但与后者相比有一些变更和补充，从而可以更好地适用于动态生成的代码，即就在执行前或执行期间在设备上编译的代码。

有关详细信息，请参阅第2-8 页的*指令集概述*。

2.2.3 ARM、Thumb 和 ThumbEE 状态

正执行 ARM 指令的处理器在 *ARM 状态*下工作。正执行 Thumb 指令的处理器在 *Thumb 状态*下工作。

在其中一种状态下工作的处理器不能执行不同指令集内的指令。例如，处于 ARM 状态下的处理器不能执行 Thumb 指令，而处于 Thumb 状态下的处理器不能执行 ARM 指令。您必须确保处理器始终不会收到与当前状态不相符的指令集的指令。

大多数 ARM 处理器始终在 ARM 状态下开始执行代码。但也有些处理器只能执行 Thumb 代码，或者可以配置为在 Thumb 状态下开始执行代码。

ThumbEE 引入了一种新的指令集状态: ThumbEE 状态。在这种状态下，根据 ThumbEE 指令集内的定义执行指令。

更改状态

每种指令集都包含用于更改处理器状态的指令。

若要在 ARM 和 Thumb 状态之间进行转换，必须切换汇编程序模式，以便使用 `ARM` 或 `THUMB` 指令生成正确的操作代码。若要生成 Thumb-2EE 代码，请使用 `THUMBX`。（使用 `CODE32` 和 `CODE16` 的汇编程序代码仍可由汇编程序进行汇编，但建议您对新代码使用 `ARM` 和 `THUMB`。）

有关详细信息，请参阅第7-60 页的*指令集和语法选择指令*。

2.2.4 处理器模式

ARM 处理器支持不同的处理器模式，具体取决于体系结构的版本（请参阅表 2-1）。

——注意——

ARMv7-M 不支持其他 ARM 处理器所采用的模式。本节不适用于 ARMv7-M。

表2-1 ARM 处理器模式

处理器模式	体系结构	模式编号
用户	全部	0b10000
FIQ - 快速中断请求	全部	0b10001
IRQ - 中断请求	全部	0b10010
超级用户	全部	0b10011
中止	全部	0b10111
未定义	全部	0b11011
系统	ARMv4 及更高版本	0b11111
监控	仅限安全扩展	0b10110

除用户模式之外，其他所有模式统称为*特权*模式。它们具有对系统资源的完全访问权限，并可随意更改模式。

需要任务保护的应用程序通常在用户模式下执行。有些嵌入式应用程序可能完全运行在超级用户模式或系统模式下。

进入除用户模式之外的模式是为了处理异常或访问特许资源（请参阅*RealView 编译工具开发指南*中的第 6 章 *处理处理器异常*）。

2.2.5 寄存器

ARM 处理器拥有 37 个寄存器。这些寄存器按部分重叠组方式加以排列。每个处理器模式都有一个不同的寄存器组。编组的寄存器为处理处理器异常和特权操作提供了快速的上下文切换。有关如何对寄存器进行编组的详细描述，请参阅*《ARM 体系结构参考手册》*。

提供了下列寄存器:

- *三十个 32 位通用寄存器*
- *程序计数器 (pc)*
- 第 2-7 页的*应用程序状态寄存器 (APSR)*
- 第 2-7 页的*保存的程序状态寄存器 (SPSR)*。

三十个 32 位通用寄存器

在任一时刻都存在十五个通用寄存器，即 r0、r1... r13、r14，具体取决于当前的处理器模式。

r13 是*堆栈指针* (sp)。C 和 C++ 编译器始终将 r13 用作堆栈指针。在 Thumb-2 中，sp 被严格定义为堆栈指针，因此如果使用 r13，则在堆栈操作中用处不大的许多指令会产生不可预测的结果。建议您不要将 sp 用作通用寄存器。

在用户模式下，r14 被用作*链接寄存器* (lr)，用于存储调用子例程时的返回地址。如果返回地址存储在堆栈上，则也可将 r14 用作通用寄存器。

在异常处理模式下，r14 存放异常的返回地址；如果在一个异常内执行子例程调用，则 r14 存放子例程的返回地址。如果返回地址存储在堆栈上，则可将 r14 用作通用寄存器。

程序计数器 (pc)

程序计数器被当作 r15（或 pc）来加以访问。它在 ARM 状态下以一个字（四字节）为增量，在 Thumb 状态下则按指令的大小执行。跳转指令将目标地址加载到 pc 中。您也可以使用数据操作指令来直接加载 PC。例如，若要从子例程返回，可以使用以下指令将链接寄存器复制到 PC 中:

```
MOV pc,lr
```

在执行期间，r15 (pc) 不包含当前执行的指令的地址。在 ARM 状态下，当前执行的指令的地址通常是 pc-8，而在 Thumb 状态下通常是 pc-4。

应用程序状态寄存器 (APSR)

APSR 存放*算术逻辑单元* (ALU) 状态标记的副本。这些标记用于确定是否执行条件指令。有关详细信息，请参阅第2-17 页的*条件执行*。

在 ARMv5TE 和 ARMv6 及更高版本中，APSR 还存放 Q 标记（请参阅第2-18 页的*ALU 状态标记*）。

在 ARMv6 及更高版本中，APSR 还存放 GE 标记（请参阅第4-99 页的*并行加法和减法*）。

可在所有模式下使用 `MSR` 和 `MRS` 指令访问这些标记。有关详细信息，请参阅第4-131 页的*MRS*和第4-133 页的*MSR*。

当前程序状态寄存器 (CPSR)

CPSR 存放下列内容:

- APSR 标记
- 当前处理器模式
- 中断禁用标记。

在支持 Thumb 或 Jazelle® 的处理器上，CPSR 还存放当前处理器状态（ARM、Thumb、ThumbEE 或 Jazelle）。

在 ARMv6T2 及更高版本中，Thumb-2 为 CPSR 引入了新的状态位。`IT` 指令使用这些位来控制 IT 块的条件执行（请参阅第4-68 页的*IT*）。

在所有模式下均可访问的标记只有 APSR 标记。对于 CPSR 的其余位，只能在特权模式下使用 `MSR` 和 `MRS` 指令访问它们。有关详细信息，请参阅第4-131 页的*MRS*和第4-133 页的*MSR*。

保存的程序状态寄存器 (SPSR)

当发生异常时，使用 SPSR 来存储 CPSR。在每种异常处理模式下，可访问一个 SPSR。用户模式和系统模式没有 SPSR，因为二者不是异常处理模式。有关详细信息，请参阅RealView 编译工具开发指南中的第 6 章 *处理处理器异常*。

2.2.6 指令集概述

所有 ARM 指令的长度都是 32 位。这些指令是按字对齐方式存储的，因此在 ARM 状态下，指令地址的两个最低有效位始终为零。

Thumb、Thumb-2 和 Thumb-2EE 指令的长度是 16 位或 32 位。这些指令按半字对齐方式存储。其中有些指令使用最低有效位来确定跳转到的目标代码是 Thumb 代码还是 ARM 代码。

在引入 Thumb-2 之前，Thumb 指令集只是 ARM 指令集功能的一个限定的子集。几乎所有 Thumb 指令都是 16 位。Thumb-2 指令集的功能与 ARM 指令集的功能几乎相同。

有关 ARM 和 Thumb 指令语法的详细信息，请参阅第 4 章 *ARM 和 Thumb 指令*。

ARM 和 Thumb 指令可划分为多个功能组：

- *跳转指令*
- *数据处理指令*
- 第2-9 页的*寄存器加载和存储指令*
- 第2-9 页的*多个寄存器加载和存储指令*
- 第2-9 页的*状态寄存器访问指令*
- 第2-9 页的*协处理器指令*。

跳转指令

此类指令用于：

- 向后跳转以构成循环
- 在条件结构中向前跳转
- 跳转到子例程
- 在 ARM 状态和 Thumb 状态之间转换处理器状态。

数据处理指令

此类指令用于对通用寄存器执行运算。它们可对两个寄存器的内容执行加法、减法或按位逻辑等运算，并将结果存放到第三个寄存器中。此外，它们还可以对单个寄存器中的值执行运算，或者对寄存器中的值与指令中提供的常数（*立即值*）执行运算。

长乘指令用两个寄存器提供 64 位的结果。

寄存器加载和存储指令

此类指令用于从内存加载单个寄存器的值，或者在内存中存储单个寄存器的值。它们可加载或存储 32 位字、16 位半字或 8 位无符号字节。可以用符号或零扩展字节和半字加载以填充 32 位寄存器。

此外，还定义了几个可将 64 位双字值加载或存储到两个 32 位寄存器的指令。

多个寄存器加载和存储指令

此类指令可从内存加载通用寄存器的任何子集，或者在内存中存储这样的子集。有关此类指令的详细描述，请参阅第2-39 页的*加载和存储多个寄存器指令*。

状态寄存器访问指令

此类指令向通用寄存器或者从通用寄存器往外移动状态寄存器的内容。

协处理器指令

此类指令支持一种用于扩展 ARM 体系结构的通用方式。

2.2.7 指令功能

本节包括以下小节:

- *条件执行*
- 第2-10 页的*寄存器访问*
- 第2-10 页的*访问内联的滚筒式移位器*。

条件执行

可以根据 APSR 中 ALU 状态标记的值，有条件地执行几乎所有 ARM 指令。虽然不需要使用跳转来跳过条件指令，但当一系列指令依赖于相同的条件时，这样做的效果会更好。

在没有 Thumb-2 的处理器上的 Thumb 状态下，条件跳转是提供条件执行的唯一机制。大多数数据处理指令会更新 ALU 标记。通常不能指定指令是否更新 ALU 标记的状态。

Thumb-2 通过使用 `IT` (If-Then) 指令和同样的 ALU 标记为条件执行提供了另一种机制。`IT` 是一个 16 位指令，最多可为后面的四个指令提供条件执行。此外，还有其他几个指令为条件执行提供了其他机制。

在 ARM 和 Thumb-2 代码中，可以指定数据处理指令是否更新 ALU 标记。可以使用一个指令所设置的标记来控制其他指令的执行，即使在它们之间有很多非标记设置指令也是如此。

有关详细描述，请参阅第2-17 页的*条件执行*。

寄存器访问

在 ARM 状态下，所有指令都可访问 r0 到 r14，并且大多数指令也可访问 r15 (pc)。`MRS` 和 `MSR` 指令可将状态寄存器的内容移到通用寄存器中，在通用寄存器中可以用普通的数据处理操作来处理这些内容。有关详细信息，请参阅第4-131 页的*MRS* 和第4-133 页的*MSR*。

Thumb-2 处理器上的 Thumb 状态提供了同样的功能，但会禁止一些对 r13 和 r15 的无用访问。

在没有 Thumb-2 的处理器上的 Thumb 状态下，大多数指令只能访问 r0 到 r7。只有少数指令能够访问 r8 到 r15。寄存器 r0 到 r7 称为低位寄存器。寄存器 r8 到 r15 称为高位寄存器。

访问内联的滚筒式移位器

ARM 算术逻辑单元具有一个 32 位滚筒式移位器，可执行移位和循环操作。对于所有 ARM 和 Thumb-2 数据处理指令和单寄存器数据传送指令的第二个操作数，可以在执行数据处理或数据传送之前，将其作为指令的一部分执行移位操作。此操作支持（但不限于）：

- 比例寻址
- 乘以一个常数
- 构造常数。

有关使用滚筒式移位器生成常数的详细信息，请参阅*第2-25 页的将常数加载到寄存器*。

Thumb-2 指令与 ARM 指令对滚筒式移位器的访问方式几乎相同。

16 位 Thumb 指令集只允许使用单独的指令来访问滚筒式移位器。

2.3 汇编语言模块的结构

汇编语言是指 ARM 汇编程序 (`armasm`) 进行分析并汇编生成对象代码的语言。缺省情况下，汇编程序应使用 ARM 汇编语言编写源代码。

`armasm` 支持用旧版本的 ARM 汇编语言编写的源代码。在这种情况下，它无需获得相应的通知。

`armasm` 还可支持用旧版本的 Thumb 汇编语言编写的源代码。在这种情况下，必须在源代码中使用 `--16` 命令行选项或 `CODE16` 指令通知 `armasm`。旧版本的 Thumb 汇编语言不支持 Thumb-2 指令。

本节介绍了以下内容:

- *汇编语言源文件的编排*
- 第2-14 页的*ARM 汇编语言模块的示例*
- 第2-16 页的 *调用子例程*。

2.3.1 汇编语言源文件的编排

汇编语言的源代码行的一般格式是:

{*label*} {*instruction*|*directive*|*pseudo-instruction*} {;*comment*}

——注意——

即使没有标签，指令、伪指令和指令前面也必须使用空格或制表符等留出空白。

————————

源代码行的所有三部分都是可选的。使用空行可使代码更具可读性。

大小写规则

指令助记符、指令和符号寄存器名称可以用大写或小写编写，但不能混合使用大小写。

行长度

为使源文件更容易阅读，可以在行尾放置反斜杠符 (\)，将较长的源代码行拆分为多个行。反斜杠后面不得有任何其他字符（包括空格和制表符）。汇编程序将反斜杠/行尾序列视为空白。

---注意---
不要在带引号的字符串内使用反斜杠/行尾序列。

行长度的最大值为 4095 个字符，包括使用反斜杠的任何扩展在内。

标签

标签是表示地址的符号。在汇编期间，将计算由标签指定的地址。

汇编程序计算标签相对于定义标签的节的原点的地址。引用同一节内的标签时可以使用 pc 加上或减去偏移量。这称为*程序相对寻址*。

其他节中标签的地址是在链接时计算的，此时链接器已在内存中为每一节分配了具体的位置。

局部标签

局部标签是标签的一个子类。局部标签以 0 到 99 范围内的一个数字开头。与其他标签不同的是，局部标签可以被定义多次。如果用宏生成标签，局部标签就十分有用。当汇编程序找到对一个局部标签的引用时，就会将其链接到该局部标签的相邻实例上。

局部标签的范围由 `AREA` 指令加以限制。使用 `ROUT` 指令可以更严格地限制其范围。

有关下列主题的详细信息，请参阅第3-26 页的*局部标签*:

- 局部标签的声明语法
- 汇编程序如何将对局部标签的引用与其标签相关联。

注释

行中的第一个分号标记注释的开始，但不包括出现在字符串常数内的分号。行的末尾就是注释的结束。一个注释本身就是一个有效的行。汇编程序将忽略所有注释。

常数

常数可以是:

数字 可接受下列形式的数字常数:

- 十进制数，例如 `123`
- 十六进制数，例如 `0x7B`
- *n_xxx*，其中:

  n 是 2 到 9 之间的基数

  xxx 是采用该基数的数字
- 浮点数，例如 0.02、123.0 或 3.14159。

仅当系统具有使用浮点数的 VFP 或 NEON 时，浮点数才可用。

布尔值 布尔常数 `TRUE` 和 `FALSE` 必须书写为 `{TRUE}` 和 `{FALSE}`。

字符 字符常数由左右单引号组成，中间括有单个字符或一个采用标准的 C 转义字符的转义字符。

字符串 字符串由用双引号括起的多个字符和空格组成。如果在一个字符串内使用了双引号或美元符号作为文本字符，则这些符号必须用一对相应的字符来表示。例如，如果需要在字符串内使用单个 `$`，则必须书写为 `$$`。在字符串常数内可以使用标准的 C 转义序列。

2.3.2 ARM 汇编语言模块的示例

示例 2-1 显示了汇编语言模块的一些核心成分。此示例是用 ARM 汇编语言编写的。在主示例目录 *install_directory*\RVDS\Examples 中以 armex.s 文件形式提供了该示例。有关如何汇编、链接和执行该示例的说明，请参阅第2-2 页的*代码示例*。

以下各节详细介绍了此示例的组成部分。

示例 2-1

```
        AREA     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; ARM semihosting (formerly SWI)

        END                     ; Mark end of file
```

ELF 节和 AREA 指令

ELF *节*是独立的、已命名的、不可分割的代码或数据序列。单个代码节是生成应用程序的最低要求。

汇编或编译的输出内容可包括:

- 一个或多个代码节。它们通常是只读节。
- 一个或多个数据节。它们通常是读写节。它们可以是*零初始化的* (ZI)。

链接器依照节位置规则，将每个节放在一个程序映像中。对于在源文件中相邻的节，在应用程序映像中不一定相邻。有关链接器如何放置节的详细信息，请参阅*《RealView 编译工具链接器和实用程序指南》*中的第 3 章 *使用基本链接器功能*。

在源文件中，AREA 指令标记一节的开始。该指令对节进行命名并设置其属性。属性放在名称后面，之间用逗号分隔。有关 AREA 指令语法的详细描述，请参阅第7-65 页的*AREA*。

可以为节选择任何名称。但是，以任何非字母字符开头的名称必须括在竖线内，否则会生成 `AREA name missing` 错误。例如，`|1_DataArea|`。

第2-14 页的示例 2-1 定义了一个名为 `ARMex` 的单个节，其中包含代码并被标记为 `READONLY`。

ENTRY 指令

`ENTRY` 指令标记的是第一个要执行的指令。在包含 C 代码的应用程序中，在 C 库初始化代码中也包含一个入口点。初始化代码和异常处理程序也包含入口点。

应用程序执行

第2-14 页的示例 2-1 中的应用程序代码在标签 `start` 处开始执行，并在此处将十进制值 10 和 3 加载到寄存器 `r0` 和 `r1` 中。这些寄存器将一起相加，并且结果将存放到 `r0` 中。

应用程序终止

在执行主代码后，应用程序会将控制权返回调试器，以此来终止执行。此操作是通过将 ARM 半主机 SVC（缺省为 `0x123456`）与下列参数结合使用来完成的：

- r0 等于 `angel_SWIreason_ReportException` (`0x18`)
- r1 等于 `ADP_Stopped_ApplicationExit` (`0x20026`)。

END 指令

此指令指示汇编程序停止处理此源文件。每个汇编语言源模块必须以仅包括 `END` 指令的一行结束。

2.3.3 调用子例程

若要调用子例程，应使用跳转和链接指令，其语法是:

```
BL  destination
```

其中，*destination* 通常是位于子例程的第一个指令处的标签。

destination 也可以是程序相对表达式。有关详细信息，请参阅第4-115 页的*B、BL、BX、BLX 和 BXJ*。

BL 指令:

- 将返回地址存放到链接寄存器中
- 将 pc 设置为子例程的地址。

在执行子例程代码后，可以使用 BX lr 指令返回。按照约定，寄存器 r0 到 r3 用于将参数传递给子例程，并且 r0 还用于将结果传递回调用方。

——注意——

单独汇编或编译的模块之间进行的调用必须符合过程调用标准规定的限制和约定。有关详细信息，请参阅 *install_directory*\Documentation\Specifications\... 中的《*ARM 体系结构的过程调用标准*》规范 aapcs.pdf。

示例 2-2 显示了一个子例程，它将两个参数值相加并将结果返回 r0。在主示例目录 *install_directory*\RVDS\Examples 中以 subrout.s 文件形式提供了该示例。有关如何汇编、链接和执行该示例的说明，请参阅第2-2 页的*代码示例*。

示例 2-2

```
        AREA    subrout, CODE, READONLY     ; Name this block of code
        ENTRY                         ; Mark first instruction to execute
start   MOV     r0, #10               ; Set up parameters
        MOV     r1, #3
        BL      doadd                 ; Call subroutine
stop    MOV     r0, #0x18             ; angel_SWIreason_ReportException
        LDR     r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SVC     #0x123456             ; ARM semihosting (formerly SWI)

doadd   ADD     r0, r0, r1            ; Subroutine code
        BX      lr                    ; Return from subroutine
        END                           ; Mark end of file
```


2.4 条件执行

在 ARM 状态下以及具有 Thumb-2 的处理器上的 Thumb 状态下，大多数数据处理指令都具有一个选项，该选项可根据运算结果来更新*应用程序状态寄存器* (APSR) 中的 ALU 状态标记。有些指令会更新所有标记，而有些指令仅更新部分标记。如果某一标记未得到更新，则会保留其原始值。每个指令的描述详细介绍了它对这些标记所具有的影响。未执行的条件指令对这些标记没有影响。

在早期体系结构中的 Thumb 状态下，大多数数据处理指令会自动更新 ALU 状态标记。没有用于更新这些标记的选项，而其他指令不能更新这些标记。

在 ARM 状态下以及具有 Thumb-2 的处理器上的 Thumb 状态下，可以根据其他指令中设置的 ALU 状态标记有条件地执行指令，执行时间为:

- 在更新这些标记的指令后立即执行
- 在尚未更新这些标记的任何数目的插入指令之后执行。

可以根据 APSR 中的 ALU 状态标记的状态有条件地执行几乎所有 ARM 指令。有关添加到指令中以使其有条件执行的后缀的列表，请参阅第2-19 页的表2-2。

在 Thumb 状态下，使用条件跳转是一种条件执行机制。

在具有 Thumb-2 的处理器上的 Thumb 状态下，可以使用特殊的 `IT` (If-Then) 指令使指令有条件地执行。此外，还可以使用 `CBZ`（零条件跳转）和 `CBNZ` 指令将寄存器值与零进行比较。

本节介绍了以下内容:

- 第2-18 页的*ALU 状态标记*
- 第2-18 页的*条件执行*
- 第2-20 页的*使用条件执行*
- 第2-21 页的*使用条件执行的示例*
- 第2-24 页的*Q 标记*。

2.4.1 ALU 状态标记

APSR 包含下列 ALU 状态标记:

N 当运算结果为负值时设置此标记。

Z 当运算结果为零时设置此标记。

C 当运算导致进位时设置此标记。

V 当运算导致溢出时设置此标记。

如果加法的结果大于或等于 2^{32}，减法的结果为正值，或者是移动或逻辑指令中的内嵌滚筒式移位器运算的结果导致进位，则会产生进位。

如果加法、减法或比较的结果大于或等于 2^{31} 或小于 -2^{31}，则会发生溢出。

2.4.2 条件执行

可有条件执行的指令具有可选条件代码，如 {*cond*} 中的语法描述所示。此条件在 ARM 指令中编码，也可在 Thumb-2 指令的前一 `IT` 指令中编码。仅当 APSR 中的条件代码标记满足指定的条件时，才会执行带有条件代码的指令。第2-19 页的表2-2 显示了可使用的条件代码。

在没有 Thumb-2 的 Thumb 处理器上，仅允许在某些跳转指令中使用 {*cond*} 字段。

表2-2 还显示了条件代码后缀与 N、Z、C 和 V 标记之间的关系。

表2-2 条件代码后缀

后缀	标记	含义
EQ	设置 Z	等于
NE	清除 Z	不等于
CS/HS	设置 C	大于或等于（无符号 >=）
CC/LO	清除 C	小于（无符号 <）
MI	设置 N	负数
PL	清除 N	正数或零
VS	设置 V	溢出
VC	清除 V	无溢出
HI	设置 C 并清除 Z	大于（无符号 >）
LS	清除 C 或设置 Z	小于或等于（无符号 <=）
GE	N 与 V 相同	有符号 >=
LT	N 与 V 不同	有符号 <
GT	清除 Z，N 与 V 相同	有符号 >
LE	设置 Z，N 与 V 不同	有符号 <=
AL	任何	始终。通常会忽略此后缀。

示例 2-3 显示了条件执行的示例。

示例 2-3

```
ADD     r0, r1, r2    ; r0 = r1 + r2, don't update flags
ADDS    r0, r1, r2    ; r0 = r1 + r2, and update flags
ADDSCS  r0, r1, r2    ; If C flag set then r0 = r1 + r2, and update flags
CMP     r0, r1        ; update flags based on r0-r1.
```

2.4.3 使用条件执行

可以利用 ARM 指令的条件执行来减少代码中跳转指令的数目。这样可提高代码密度。Thumb-2 中的 `IT` 指令也实现了类似的改进。

跳转指令在处理器周期中也是很耗时的。在没有跳转预测硬件的 ARM 处理器上，每执行一次跳转，通常就需要三个处理器周期来重填处理器管道。

有些 ARM 处理器（如 ARM10™ 和 StrongARM®）具有跳转预测硬件。在使用这些处理器的系统中，仅当存在误预测时才需要刷新和重填管道。

2.4.4 使用条件执行的示例

此示例使用*最大公约数* (gcd) 算法 (Euclid) 的两种实现方法。它演示了如何使用条件执行来提高代码密度和执行速度。有关执行速度的详细分析仅适用于 ARM7™ 处理器。代码密度计算则适用于所有 ARM 处理器。

在 C 语言中，该算法可以表示为:

```
int gcd(int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

采用以下方法，可以只用跳转条件执行来实现 gcd 函数:

```
gcd     CMP      r0, r1
        BEQ      end
        BLT      less
        SUBS     r0, r0, r1  ; could be SUB r0, r0, r1 for ARM
        B        gcd
less
        SUBS     r1, r1, r0  ; could be SUB r1, r1, r0 for ARM
        B        gcd
end
```

由于跳转数目的限制，该代码的长度是七个指令。每执行一次跳转，处理器就必须重填管道并从新位置继续执行。其他指令和未执行的跳转各使用一个周期。

通过使用 ARM 指令集的条件执行功能，仅用四个指令即可执行 gcd 函数:

```
gcd
        CMP      r0, r1
        SUBGT    r0, r0, r1
        SUBLE    r1, r1, r0
        BNE      gcd
```

除了缩短代码长短之外，大多数情况下此代码的执行速度也比较快。表2-3 和表 2-4 在 r0 等于 1 且 r1 等于 2 的情况下显示了每种执行方法所使用的周期数目。在这种情况下，用有条件地执行所有指令来代替跳转可节省三个周期。

在 r0 等于 r1 时的任何情况下，两种形式的代码的执行周期数都相等。在其他所有情况下，条件形式的代码的执行周期数较少。

表2-3 仅使用条件跳转

r0: a	r1: b	指令	周期数 (ARM7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1（未执行）
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			总计 = 13

表2-4 所有指令都是条件指令

r0: a	r1: b	指令	周期数 (ARM7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1（未执行）
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1（未执行）
1	1	`SUBLT r1,r1,r0`	1（未执行）
1	1	`BNE gcd`	1（未执行）
			总计 = 10

gcd 的 16 位 Thumb 版本

由于 `B` 是可以有条件执行的唯一的 16 位 Thumb 指令，因此必须用 Thumb 代码中的条件跳转来编写 gcd 算法。

与 ARM 条件跳转执行方法类似，Thumb 代码也需要七个指令。在使用 Thumb 指令时，与较小的 ARM 代码执行的 16 字节相比，整个代码大小只有 14 字节。

此外，在使用 16 位内存的系统中，Thumb 版本比第二种 ARM 执行方法运行得*快*，因为每个 16 位 Thumb 指令只需要访问一次内存，而每个 ARM 32 位指令需要两次存取。

gcd 的 Thumb-2 版本

通过使用 `IT` 指令使 `SUB` 成为条件指令，可以将此代码的 ARM 版本转换为 Thumb-2 代码：

```
gcd
        CMP     r0, r1
        ITE     GT
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

上述代码会正常地同样汇编为 ARM 或 Thumb-2 代码。汇编程序会检查 `IT` 指令，但在汇编为 ARM 代码时会忽略这些指令。（您可以忽略 IT 指令。在汇编为 Thumb-2 代码时汇编程序会插入此类指令。）

如果 Thumb-2 代码中所需的指令比 ARM 代码多一个，则整个代码大小在 Thumb-2 代码中为 10 字节，而在 ARM 代码中则为 16 字节。

执行速度

若要优化代码的执行速度，您必须具备指令时序、跳转预测逻辑以及目标系统的缓存行为等方面的详细知识。有关完整的信息，请参阅*《ARM 体系结构参考手册》*和各处理器的技术参考手册。

2.4.5 Q 标记

在 ARMv5TE 和 ARMv6 及更高版本中，当饱和算术指令（请参阅第4-94 页的*QADD、QSUB、QDADD 和 QDSUB*）中出现饱和或某些乘法指令（请参阅第4-77 页的*SMULxy 和 SMLAxy* 和第4-79 页的*SMULWy 和 SMLAWy*）中出现溢出时，会记录 Q 标记。

Q 标记是一种*粘性*标记。虽然这些指令可以设置该标记，但不能清除它。您可以执行一系列这种指令，然后测试该标记，以确定是否在指令系列中的任何点发生了饱和或溢出，而无需在每个指令之后检查该标记。

要清除 Q 标记，则请使用 `MSR` 指令（请参阅第4-133 页的*MSR*）。

不能直接用条件代码来测试 Q 标记的状态。若要读取 Q 标记的状态，请使用 `MRS` 指令（请参阅第4-131 页的*MRS*）。

2.5 将常数加载到寄存器

只有在执行从内存加载数据时，才能用单个指令将任何 32 位立即数加载到寄存器中。这是因为 ARM 和 Thumb-2 指令的长度仅是 32 位。在使用 16 位 Thumb 指令时，可生成的常数范围要小得多。

此外，也可以直接将许多常用的常数作为操作数包含在数据处理指令内，而无需单独加载它们。

可以用一次数据加载来将任何 32 位值加载到寄存器中，但也可以使用其他更直接、更有效的方式来加载许多常用的常数。

在 ARMv6T2 及更高版本中，还可以先后用 `MOV` 和 `MOVT` 这两个指令将任何 32 位值加载到寄存器中。可以使用 `MOV32` 伪指令来构建指令序列。

以下各节介绍:

- 如何使用 `MOV` 和 `MVN` 指令加载一系列立即值。

  请参阅第2-26 页的*用 MOV 和 MVN 直接加载*。

- 如何使用 `MOV32` 伪指令加载任何 32 位常数。

  请参阅第2-30 页的*用 MOV32 加载*。

- 如何使用 `LDR` 伪指令加载任何 32 位常数。

  请参阅第2-30 页的*用 LDR Rd, =const 加载*。

- 如何加载浮点常数。

  请参阅第2-32 页的*加载浮点常数*。

2.5.1 用 MOV 和 MVN 直接加载

在 ARM 和 Thumb-2 中，可以使用 32 位 `MOV` 和 `MVN` 指令直接将一系列常数值加载到寄存器中。

16 位 Thumb `MOV` 指令可以加载位于范围 0 到 255 内的任何常数。不能使用 16 位 `MVN` 指令加载常数。

ARM 状态立即数 显示了 ARM 中的单个指令可加载的值的范围。第2-28 页的 *Thumb-2 立即数* 显示了 Thumb-2 中的单个指令可加载的值的范围。

您无需决定是使用 `MOV` 还是 `MVN`。汇编器会使用适当的指令。如果值是一个汇编时的变量，则这一点十分有用。

如果所编写的指令包含不可用的常数，汇编程序就会报告以下错误：`Immediate n out of range for this operation.`

ARM 状态立即数

在 ARM 状态下：

- 在范围 `0x0-0xFF` (0-255) 内，`MOV` 可加载任何 8 位常数值。

  该指令还可以将这些值循环移动任何偶数位。

  在许多数据处理操作中，这些值也可用作立即操作数，而无需用单独的指令加载。
- `MVN` 可加载这些值的按位补码。这些补码数值为 `-(n+1)`，其中 *n* 是 `MOV` 中给出的值。
- 在 ARMv6T2 及更高版本中，`MOV` 可以在 `0x0-0xFFFF` (0-65535) 范围内加载任何 16 位数字。

第2-27 页的表2-5 显示了此指令所提供的 8 位值的范围（用于数据处理操作）。

第2-27 页的表2-6 显示了此指令所提供的 16 位值的范围（仅适用于 `MOV` 指令）。

表2-5 ARM 状态立即数（8 位）

二进制	十进制	乘方	十六进制	MVN 值[a]	注释
`000000000000000000000000abcdefgh`	0-255	1	`0-0xFF`	-1 到 -256	-
`0000000000000000000000abcdefgh00`	0-1020	4	`0-0x3FC`	-4 到 -1024	-
`00000000000000000000abcdefgh0000`	0-4080	16	`0-0xFF0`	-16 到 -4096	-
`000000000000000000abcdefgh000000`	0-16320	64	`0-0x3FC0`	-64 到 -16384	-
	...	...	...	...	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0-0xFF000000`	1-256 x -2^{24}	-
`cdefgh000000000000000000000000ab`	（位模式）	-	-	（位模式）	请参阅*注释*中的 b
`efgh000000000000000000000000abcd`	（位模式）	-	-	（位模式）	请参阅*注释*中的 b
`gh000000000000000000000000abcdef`	（位模式）	-	-	（位模式）	请参阅*注释*中的 b

表2-6 MOV 指令中的 ARM 状态立即数

二进制	十进制	乘方	十六进制	MVN 值	注释
`0000000000000000abcdefghijklmnop`	0-65535	1	`0-0xFFFF`	-	请参阅*注释*中的 c

注释

这些注释提供了有关表2-5 和表2-6 的附加信息。

a `MVN` 值不能直接用作其他非数据处理指令中的操作数。

b 这些值只能在 ARM 状态下使用。除了特别标注的值之外，此表中的所有其他值还可在 Thumb-2 中使用。

c 这些值只能在 ARMv6T2 及更高版本中使用。它们不能直接用作其他指令中的操作数。

Thumb-2 立即数

在 ARMv6T2 及更高版本中的 Thumb 状态下：

- 32 位 `MOV` 指令可加载：
  - — `0x0-0xFF` (0-255) 范围内的任何 8 位常数值。
  - — 左移任何数字的任何 8 位常数
  - — 在所有四字节寄存器中复制的任何 8 位位模式
  - — 在字节 0 和 2（字节 1 和 3 设置为 0）中复制的任何 8 位位模式。
  - — 在字节 1 和 3（字节 0 和 2 设置为 0）中复制的任何 8 位位模式。

  在许多数据处理操作中，这些值也可用作立即操作数，而无需用单独的指令加载。

- 32 位 `MVN` 指令可以加载这些值的按位补码。这些补码数值为 `-(n+1)`，其中 *n* 是 `MOV` 中给出的值。

- 32 位 `MOV` 指令可以加载 `0x0-0xFFFF` (0-65535) 范围内的任何 16 位数字。这些值不能直接用作数据处理操作中的立即操作数。

第2-29 页的表2-7 显示了此指令所提供的值的范围（用于数据处理操作）。

第2-29 页的表2-8 显示了此指令所提供的 16 位值的范围（仅适用于 `MOV` 指令）。

表2-7 Thumb 状态立即数

二进制	十进制	乘方	十六进制	MVN 值[a]	注释
`000000000000000000000000abcdefgh`	0-255	1	`0-0xFF`	-1 到 -256	-
`00000000000000000000000abcdefgh0`	0-510	2	`0-0x1FE`	-2 到 -512	-
`0000000000000000000000abcdefgh00`	0-1020	4	`0-0x3FC`	-4 到 -1024	-
	...	...	...	...	-
`0abcdefgh00000000000000000000000`	0-255 x 2^{23}	2^{23}	`0-0x7F800000`	1-256 x -2^{23}	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0-0xFF000000`	1-256 x -2^{24}	-
`abcdefghabcdefghabcdefghabcdefgh`	（位模式）	-	`0xXYXYXYXY`	`0xXYXYXYXY`	-
`00000000abcdefgh00000000abcdefgh`	（位模式）	-	`0x00XY00XY`	`0xFFXYFFXY`	-
`abcdefgh00000000abcdefgh00000000`	（位模式）	-	`0xXY00XY00`	`0xXYFFXYFF`	-
`00000000000000000000abcdefghijkl`	0-4095	1	`0-0xFFF`	-	请参阅*注释*中的 b

表2-8 MOV 指令中的 Thumb 状态立即数

二进制	十进制	乘方	十六进制	MVN 值	注释
`0000000000000000abcdefghijklmnop`	0-65535	1	`0-0xFFFF`	-	请参阅*注释*中的 c

注释

这些注释提供了有关表2-7 和表2-8 的附加信息。

a `MVN` 值不能直接用作其他指令中的操作数。

b 这些值可直接用作 `ADD`、`SUB` 和 `MOV` 指令中的操作数，但不能用于 `MVN` 或其他任何数据处理指令中的操作数。

c 这些值只能在 `MOV` 指令中使用。

2.5.2 用 MOV32 加载

在 ARMv6T2 中，ARM 和 Thumb-2 指令集均包含：

- 一个 MOV 指令，它可以将位于范围 0x00000000 到 0x0000FFFF 内的任何值加载到寄存器中
- 一个 MOVT 指令，无需更改最低有效半部的内容，它就可将位于范围 0x0000 到 0xFFFF 内的任何值加载到寄存器的最高有效半部。

可使用这两个指令在寄存器中构造任何 32 位常数。也可使用 MOV32 伪指令。汇编程序会自动生成 MOV 和 MOVT 指令对。有关 MOV32 伪指令语法的描述，请参阅第4-151 页的*MOV32 伪指令*。

2.5.3 用 LDR Rd, =const 加载

LDR Rd,=*const* 伪指令可在单个指令中构造任何 32 位数字常数。使用此伪指令可生成超出 MOV 和 MVN 指令允许范围的常数。

LDR 伪指令可为特定的常数生成最高效的单个指令：

- 如果可以用 MOV 或 MVN 指令构造该常数，则汇编程序会生成适当的指令。
- 如果不能用 MOV 或 MVN 指令构造该常数，则汇编程序会执行下列操作：
  - — 将该值放入*文字池*（在代码中嵌入的一部分内存，用于存放常数值）中
  - — 生成一个使用程序相对的寻址的 LDR 指令，用于从文字池中读取该常数。

  例如：

```
LDR      rn, [pc, #offset to literal pool]
                        ; load register n with one word
                        ; from the address [pc + offset]
```

  必须确保文字池位于汇编程序所生成的 LDR 指令的范围内。有关详细信息，请参阅第2-30 页的*放置文字池*。

有关 LDR 伪指令语法的描述，请参阅第4-153 页的*LDR 伪指令*。

放置文字池

汇编程序将文字池放在每节的末尾。这些节的末尾是由下一节开始处的 AREA 指令定义的，或者是由汇编代码末尾的 END 指令定义的。位于包含文件末尾的 END 指令并不表示一节的结束。

在较大的节中，缺省的文字池可能会超出一个或多个 LDR 指令的范围。从 pc 到常数的偏移量必须符合下列条件：

- 在 ARM 或 Thumb-2 代码中小于 4KB，但可指向任何方向
- 使用 16 位指令时，在 Thumb 代码中小于 1KB 并指向前面。

当 LDR Rd,=const 伪指令要求将常数放入文字池时，汇编程序会执行下列操作：

- 检查以前任何文字池中的常数是否可用以及是否可寻址。如果是，则会对现有常数进行寻址。
- 如果以前的文字池已经不可用，则会尝试将常数放入下一个文字池中。

如果下一个文字池超出范围，汇编程序会生成一条错误消息。在这种情况下，必须使用 LTORG 指令在代码中放置一个附加的文字池。LTORG 指令应放在失败的 LDR 伪指令之后，并位于范围 ±4KB（ARM，32 位 Thumb-2）或范围 0 到 +1KB (Thumb、16 位 Thumb-2）内。有关详细描述，请参阅第7-18 页的*LTORG*。

文字池必须放在处理器不会试图将其当作指令来执行的位置上。它们应放在无条件跳转指令的后面，或者放在子例程末尾处的返回指令的后面。

示例 2-4 显示了这一操作原理。在主示例目录 *install_directory*\RVDS\Examples 中以 loadcon.s 文件形式提供了该示例。有关如何汇编、链接和执行该示例的说明，请参阅第2-2 页的*代码示例*。

带有注释的指令是汇编程序生成的 ARM 指令。

示例 2-4

```
        AREA     Loadcon, CODE, READONLY
        ENTRY                                ; Mark first instruction to execute
start
        BL       func1                       ; Branch to first subroutine
        BL       func2                       ; Branch to second subroutine
stop
        MOV      r0, #0x18                   ; angel_SWIreason_ReportException
        LDR      r1, =0x20026                ; ADP_Stopped_ApplicationExit
        SVC      #0x123456                   ; ARM semihosting (formerly SWI)

func1
        LDR      r0, =42                     ; => MOV R0, #42
        LDR      r1, =0x55555555             ; => LDR R1, [PC, #offset to
                                             ; Literal Pool 1]
        LDR      r2, =0xFFFFFFFF             ; => MVN R2, #0
        BX       lr
        LTORG                                ; Literal Pool 1 contains
```



```
                                            ; literal 0x55555555
func2
        LDR      r3, =0x55555555           ; => LDR R3, [PC, #offset to
                                            ; Literal Pool 1]
        ; LDR r4, =0x66666666              ; If this is uncommented it
                                            ; fails, because Literal Pool 2
                                            ; is out of reach
        BX       lr
LargeTable
        SPACE    4200                       ; Starting at the current location,
                                            ; clears a 4200 byte area of memory
                                            ; to zero
        END                                 ; Literal Pool 2 is empty
```

2.5.4 加载浮点常数

在 NEON 和 VFPv3 指令集内，有一些指令可将有限范围的浮点常数作为立即数加载。请参阅：

- 第5-37 页的*VMOV*、*VMVN（立即数）*，可了解 NEON 指令的详细信息
- 第5-96 页的*VMOV*，可了解 VFPv3 指令的详细信息。

可使用 `VLDR` 伪指令将任何单精度或双精度浮点值从文字池加载到单个指令中。

有关详细信息，请参阅第5-77 页的*VLDR 伪指令*。

2.6 将地址加载到寄存器中

通常需要将地址加到寄存器中。可能需要加载变量、字符串常数或跳转表的起始位置的地址。

地址通常表示为相对当前 pc 或其他寄存器的偏移量。

本节描述了下列将地址加载到寄存器中的方法:

- 直接加载到寄存器，请参阅*用 ADR 和 ADRL 直接加载*
- 从文字池加载地址，请参阅第2-36 页的*用 LDR Rd, =label 加载地址*。

2.6.1 用 ADR 和 ADRL 直接加载

利用 `ADR` 指令和 `ADRL` 伪指令，无需执行数据加载即可生成位于某一范围内的地址。`ADR` 和 `ADRL` 接受程序相对表达式，这是一个带有可选偏移量的标签，其中标签地址是相对于当前 pc 的。

——注意——

用于 `ADR` 或 `ADRL` 的标签必须位于同一代码节中。如果同一节中对标签的引用超出范围，汇编程序就会产生错误。

在 Thumb 状态下，16 位 `ADR` 指令只能生成字对齐的地址。

在没有 Thumb-2 的处理器上的 Thumb 状态下，`ADRL` 将不可用。

ADR

可用范围因指令集而异:

ARM	±255 到字节或半字对齐的地址。
	±1020 字节到字对齐的地址。
32 位 Thumb-2	±4095 字节到字节、半字或字对齐的地址。
16 位 Thumb	0 到 1020 字节。`label` 必须为字对齐。可以使用 `ALIGN` 指令确保这一点。

有关详细信息，请参阅第4-22 页的*ADR*。

ADRL

汇编程序通过下列方式转换 ADRL *rn*,*label* 伪指令:

- 如果地址位于有效范围内，则生成两个数据处理指令来加载地址
- 如果不能在两个指令中构造地址，则生成一条错误消息。

可用范围取决于所用的指令集:

ARM	±64KB 到字节或半字对齐的地址。 ±256KB 到字对齐的地址。
32 位 Thumb-2	±1MB 到字节、半字或字对齐的地址。
16 位 Thumb	ADRL 不可用。

有关加载超出 ADRL 伪指令范围的地址的信息，请参阅第2-36 页的*用 LDR Rd, =label 加载地址*。

用 ADR 执行跳转表

示例 2-5 显示了执行跳转表的 ARM 代码。在该示例中，ADR 指令将加载跳转表的地址。在主示例目录 *install_directory*\RVDS\Examples 中以 jump.s 文件形式提供了该示例。有关如何汇编、链接和执行该示例的说明，请参阅第2-2 页的*代码示例*。

示例 2-5 执行跳转表 (ARM)

```
        AREA    Jump, CODE, READONLY      ; Name this block of code
        ARM                               ; Following code is ARM code
num     EQU     2                         ; Number of entries in jump table
        ENTRY                             ; Mark first instruction to execute
start                                     ; First instruction to call
        MOV     r0, #0                    ; Set up the three parameters
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc                 ; Call the function
stop
        MOV     r0, #0x18                 ; angel_SWIreason_ReportException
        LDR     r1, =0x20026              ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                 ; ARM semihosting (formerly SWI)
arithfunc                                 ; Label the function
        CMP     r0, #num                  ; Treat function code as unsigned
integer
        BXHS    lr                        ; If code is >= num then simply return
        ADR     r3, JumpTable             ; Load address of jump table
```



```
        LDR     pc, [r3,r0,LSL#2]       ; Jump to the appropriate routine
JumpTable
        DCD     DoAdd
        DCD     DoSub

DoAdd
        ADD     r0, r1, r2              ; Operation 0
        BX      lr                      ; Return
DoSub
        SUB     r0, r1, r2              ; Operation 1
        BX      lr                      ; Return
        END                             ; Mark the end of this file
```

在第2-34 页的示例 2-5 中，函数 arithfunc 采用三个参数，并将结果返回到 r0。第一个自变量确定要对第二个和第三个自变量执行的运算：

参数 1 = 0 结果 = 参数 2 + 参数 3。

参数 1 = 1 结果 = 参数 2 - 参数 3。

跳转表通过下列指令和汇编程序指令来执行：

EQU 是一个汇编程序指令，用于为某一符号赋值。在第2-34 页的示例 2-5 中，它将值 2 赋予 *num*。当在代码中的其他位置使用 *num* 时，将替换值 2。以这种方式使用 EQU 与在 C 语言中使用 #define 来定义常数类似。

DCD 声明一个或多个存储字。在第2-34 页的示例 2-5 中，每个 DCD 分别存储一个例程的地址，而每个例程处理跳转表的一个特定子句。

LDR LDR pc,[r3,r0,LSL#2] 指令将跳转表所需子句的地址加载到 pc 中，并执行下列操作：

- 将 r0 中的子句数值乘以 4 以给出一个字偏移量
- 将结果加到跳转表的地址中
- 将组合地址的内容加载到 pc 中。

2.6.2 用 LDR Rd, =label 加载地址

`LDR Rd,=` 伪指令可将任何 32 位常数加载到寄存器中（请参阅第2-30 页的*用 LDR Rd, =const 加载*）。此外，它还接受程序相对表达式，如标签以及带偏移量的标签。

汇编程序通过下列方式转换 `LDR r0, =label` 伪指令：

- 将 *label* 的地址放入文字池（在代码中嵌入的一部分内存，用于存放常数值）。
- 生成程序相对的 `LDR` 指令，以便从文字池读取该地址，例如：

```
LDR     rn [pc, #offset to literal pool]
                        ; load register n with one word
                        ; from the address [pc + offset]
```

  必须确保在指定范围内有文字池（有关详细信息，请参阅第2-30 页的*放置文字池*）。

与 `ADR` 和 `ADRL` 伪指令不同的是，可以对当前节之外的标签使用 `LDR`。如果标签超出当前节范围，则在汇编源文件时，汇编程序会在对象代码中放置一个重新定位指令。重新定位指令指示链接器在链接时解析该地址。无论链接器将包含 `LDR` 和文字池的节放在何处，该地址都保持有效。

示例 2-6 显示了这一操作原理。在主示例目录 *install_directory*`\RVDS\Examples` 中以 `ldrlabel.s` 文件形式提供了该示例。有关如何汇编、链接和执行该示例的说明，请参阅第2-2 页的*代码示例*。

带有注释的指令是由汇编程序生成的 ARM 指令。

示例 2-6

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                           ; Mark first instruction to execute
start
        BL      func1                   ; Branch to first subroutine
        BL      func2                   ; Branch to second subroutine
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting (formerly SWI)
func1
        LDR     r0, =start              ; => LDR R0,[PC, #offset into
                                        ; Literal Pool 1]
        LDR     r1, =Darea + 12         ; => LDR R1,[PC, #offset into
                                        ; Literal Pool 1]
```

```
        LDR     r2, =Darea + 6000          ; => LDR R2, [PC, #offset into
                                           ; Literal Pool 1]
        BX      lr                         ; Return
        LTORG                              ; Literal Pool 1
func2
        LDR     r3, =Darea + 6000          ; => LDR r3, [PC, #offset into
                                           ; Literal Pool 1]
                                           ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004          ; If uncommented produces an error
                                           ; as Literal Pool 2 is out of range
        BX      lr                         ; Return
Darea   SPACE   8000                       ; Starting at the current location,
                                           ; clears a 8000 byte area of memory
                                           ; to zero
        END                                ; Literal Pool 2 is out of range of
                                           ; the LDR instructions above
```


LDR Rd, =label 示例：字符串复制

示例 2-7 显示了用一个字符串覆盖另一个字符串的 ARM 代码例程。该例程使用 LDR 伪指令从一个数据节中加载两个字符串的地址。有以下几点需要特别注意：

DCB　　DCB 指令定义一个或多个存储字节。除了整数值之外，DCB 还接受带引号的字符串。字符串的每个字符均存放在连续的字节中。有关详细信息，请参阅第7-22 页的*DCB*。

LDR, STR　　LDR 和 STR 指令使用后变址寻址来更新其地址寄存器。例如，指令：

```
LDRB    r2,[r1],#1
```

用 r1 所指向的地址的内容加载 r2，然后将 r1 增加 1。

示例 2-7 字符串复制

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                          ; Mark first instruction to execute
start
        LDR     r1, =srcstr            ; Pointer to first string
        LDR     r0, =dststr            ; Pointer to second string
        BL      strcopy                ; Call subroutine to do copy
stop
        MOV     r0, #0x18              ; angel_SWIreason_ReportException
        LDR     r1, =0x20026           ; ADP_Stopped_ApplicationExit
        SVC     #0x123456              ; ARM semihosting (formerly SWI)
strcopy
        LDRB    r2, [r1],#1            ; Load byte and update address
        STRB    r2, [r0],#1            ; Store byte and update address
        CMP     r2, #0                 ; Check for zero terminator
        BNE     strcopy                ; Keep going if not
        MOV     pc,lr                  ; Return

        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```


2.7 加载和存储多个寄存器指令

ARM、Thumb 和 Thumb-2 指令集包含用于从内存加载和在内存中存储多个寄存器的指令。

多个寄存器传送指令提供了一种高效的方法，可向内存以及从内存往外移动多个寄存器的内容。它们最常用于块复制以及子例程入口点和退出点处的堆栈操作。使用多个寄存器传送指令来代替一系列单个数据传送指令具有很多优点，包括:

- 降低了代码大小。
- 只需一次指令存取开销，而不是很多次指令存取开销。
- 在不带高速缓存的 ARM 处理器上，由加载多个或存储多个指令传送的数据的第一个字始终是无序内存周期，但随后传送的所有字都是有序内存周期。在大多数系统中，有序内存周期更快一些。

——注意——

编号最低的寄存器在所访问的最低内存地址之间传送，编号最高的寄存器在所访问的最高内存地址之间传送。对于指令中寄存器列表内的寄存器，其顺序则没有什么区别。

使用 `--diag_warning 1206` 汇编程序命令行选项，可以检查寄存器列表中的寄存器是否是按升序指定的。

本节介绍了以下内容:

- 第2-40 页的*加载和存储可在 ARM 和 Thumb 中使用的多个指令*
- 第2-41 页的*用 LDM 和 STM 执行堆栈*
- 第2-43 页的*用 LDM 和 STM 执行块复制*。

2.7.1 加载和存储可在 ARM 和 Thumb 中使用的多个指令

下列指令都可在 ARM 和 Thumb 指令集内使用:

`LDM`	加载多个寄存器。
`STM`	存储多个寄存器。
`PUSH`	将多个寄存器存储到堆栈中并更新堆栈指针。
`POP`	从堆栈中加载多个寄存器，并更新堆栈指针。

在 `LDM` 和 `STM` 指令中:

- 对于所加载或存储的寄存器的列表:
  - — 在 ARM 指令中，列表可包含 r0 到 r15 中的全部或任何一些
  - — 在 32 位 Thumb-2 指令中，列表可包含 r0 到 r12 中的全部或任何一些；在有些指令中，还可包含 r14 或 r15
  - — 在 16 位 Thumb 和 Thumb-2 指令中，列表可包含 r0 到 r7 中的全部或任何一些。
- 地址可以:
  - — 在每次传送后递增
  - — 在每次传送前递增（仅限 ARM 指令）
  - — 在每次传送后递减（仅限 ARM 指令）
  - — 在每次传送前递减（不包括 16 位 Thumb）。
- 基址寄存器可以:
  - — 更新为内存中的下一数据块
  - — 按执行指令之前的样子保留（不包括 16 位 Thumb）。

在基址寄存器更新为指向内存中的下一个块时，这称为*回写*，即调整后的地址将写回基址寄存器。

在 `PUSH` 和 `POP` 指令中:

- 堆栈指针 (r13) 是基址寄存器，并始终会进行更新。
- 每次在 `POP` 指令中执行传送后递增地址，每次在 `PUSH` 指令中执行传送前递减地址。

- 对于所加载或存储的寄存器的列表:
  - 在 ARM 指令中，列表可包含 r0 到 r15 中的全部或任何一些
  - 在 32 位 Thumb-2 指令中，列表可包含 r0 到 r12 中的全部或任何一些；在有些指令中，还可包含 r14 或 r15
  - 在 16 位 Thumb-2 和 Thumb 指令中，列表可包含 r0 到 r7 中的全部或任何一些；并可包含 r14（仅限 PUSH）或 r15（仅限 POP）。

2.7.2 用 LDM 和 STM 执行堆栈

加载和存储多个指令可以更新基址寄存器。对于堆栈操作来说，基址寄存器通常是堆栈指针 r13。这就意味着，可以在单个指令中使用这些指令对任何数量的寄存器执行推入和弹出操作。

加载和存储多个指令可用于多种类型的堆栈:

降序或升序

堆栈向下增长或向上增长，前者从一个高地址开始并向更低地址前进（*降序*堆栈），而后者从一个低地址开始并向更高地址前进（*升序*堆栈）。

满或空

堆栈指针可指向堆栈中的最后一项（*满*堆栈），也可指向堆栈中的下一个空闲空间（*空*堆栈）。

为了给程序员提供方便，可使用面向堆栈的后缀来替代原有后缀之前或之后的增量或减量。有关面向堆栈的后缀列表，请参阅表2-9。

表2-9 用于加载和存储多个指令的后缀

堆栈类型	推入	弹出
满降序	STMFD（STMDB，之前递减）	LDMFD（LDM，之后递增）
满升序	STMFA（STMIB，之前递增）	LDMFA（LDMDA，之后递减）
空降序	STMED（STMDA，之后递减）	LDMED（LDMIB，之前递增）
空升序	STMEA（STM，之后递增）	LDMEA（LDMDB，之前递减）

例如:

```
STMFD    r13!, {r0-r5}  ; Push onto a Full Descending Stack
LDMFD    r13!, {r0-r5}  ; Pop from a Full Descending Stack
```


———注意———

《ARM 体系结构的过程调用标准》(AAPCS) 以及 ARM 和 Thumb C 和 C++ 编译器始终使用满降序堆栈。

PUSH 和 POP 指令采用满降序堆栈。它们是使用回写的 STMDB 和 LDM 的首选同义词。

———

用于嵌套子例程的堆栈寄存器

在子例程入口点和退出点处，堆栈操作非常有用。在一个子例程的开始处，可以将所需的任何工作寄存器存储在堆栈上，然后在退出时将其再次弹出。

此外，如果在入口点处将链接寄存器推入堆栈，就可以安全地进行其他子例程调用，而不会导致返回地址丢失。这样，还可以在退出时将 pc 弹出堆栈，从而从子例程返回，而不必先弹出 lr 然后再将该值移到 pc 中。例如：

```
subroutine  PUSH    {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            POP     {r5-r7,pc} ; Pop work registers and pc
```

———注意———

在混合使用 ARM 和 Thumb 的系统中要慎用此方法。在 ARMv4T 系统中，不能通过直接弹出到 pc 来更改状态。在这些情况下，必须将地址弹入临时寄存器并使用 BX 指令。

在 ARMv5T 及更高版本中，可以用这种方式来更改状态。

有关混合使用 ARM 和 Thumb 的详细信息，请参阅RealView 编译工具开发指南中的第 4 章 *交互操作 ARM 和 Thumb*。

———

2.7.3 用 LDM 和 STM 执行块复制

示例 2-8 是一个 ARM 代码例程，它通过一次复制单个字，将一组字从源位置复制到目标位置中。在主示例目录 *install_directory*\RVDS\Examples 中以 word.s 文件形式提供了该示例。有关如何汇编、链接和执行该示例的说明，请参阅第2-2 页的*代码示例*。

示例 2-8 在不使用 LDM 和 STM 的情况下执行块复制

```
        AREA    Word, CODE, READONLY    ; name this block of code
num     EQU     20                      ; set number of words to be copied
        ENTRY                           ; mark the first instruction called
start
        LDR     r0, =src                ; r0 = pointer to source block
        LDR     r1, =dst                ; r1 = pointer to destination
block
        MOV     r2, #num                ; r2 = number of words to copy
wordcopy
        LDR     r3, [r0], #4            ; load a word from the source and
        STR     r3, [r1], #4            ; store it to the destination
        SUBS    r2, r2, #1              ; decrement the counter
        BNE     wordcopy                ; ... copy more
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting (formerly SWI)

        AREA    BlockData, DATA, READWRITE
src     DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        END
```

此模块通过使用 LDM 和 STM 尽可能多地复制内容，提高了效率。在 ARM 所拥有的寄存器数目一定的情况下，一次传送八个字是比较合理的。利用以下指令，可以求出要复制的块所包含的八个字的倍数（如果 r2 = 要复制的字数）：

```
MOVS   r3, r2, LSR #3    ; number of eight word multiples
```

此值可用于控制循环中的迭代次数，每次迭代复制八个字。如果剩余的字不足八个时，可以用以下指令求出剩余的字数（假定 r2 尚未损坏）：

```
ANDS   r2, r2, #7
```

第2-44 页的示例 2-9 列出了为使用 LDM 和 STM 执行复制而重新写入的块复制模块。

示例 2-9 使用 LDM 和 STM 执行块复制

```
            AREA    Block, CODE, READONLY    ; name this block of code
num         EQU     20                       ; set number of words to be copied
            ENTRY                            ; mark the first instruction called
start
            LDR     r0, =src                 ; r0 = pointer to source block
            LDR     r1, =dst                 ; r1 = pointer to destination block
            MOV     r2, #num                 ; r2 = number of words to copy
            MOV     sp, #0x400               ; Set up stack pointer (r13)
blockcopy
            MOVS    r3,r2, LSR #3            ; Number of eight word multiples
            BEQ     copywords                ; Less than eight words to move?
            PUSH    {r4-r11}                 ; Save some working registers
octcopy
            LDM     r0!, {r4-r11}            ; Load 8 words from the source
            STM     r1!, {r4-r11}            ; and put them at the destination
            SUBS    r3, r3, #1               ; Decrement the counter
            BNE     octcopy                  ; ... copy more
            POP     {r4-r11}                 ; Don't need these now - restore
                                             ; originals
copywords
            ANDS    r2, r2, #7               ; Number of odd words to copy
            BEQ     stop                     ; No words left to copy?
wordcopy
            LDR     r3, [r0], #4             ; Load a word from the source and
            STR     r3, [r1], #4             ; store it to the destination
            SUBS    r2, r2, #1               ; Decrement the counter
            BNE     wordcopy                 ; ... copy more
stop
            MOV     r0, #0x18                ; angel_SWIreason_ReportException
            LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
            SVC     #0x123456                ; ARM semihosting (formerly SWI)

            AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```


2.8 使用宏

宏定义是位于 `MACRO` 和 `MEND` 指令之间的代码块。它定义了一个名称，可用于代替重复整个代码块。宏的主要用途是:

- 通过用有意义的单个名称来代替代码块，使用户可以更容易弄清楚源代码的逻辑
- 避免多次重复一个代码块。

有关详细信息，请参阅第 7-32 页的*MACRO 和 MEND*。

本节介绍了以下内容:

2.8.1 测试并跳转宏示例

在没有 Thumb-2 的处理器上的 ARM 和 Thumb 代码中，测试并跳转操作需要执行两个 ARM 指令。

可以定义一个与下面类似的宏定义:

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc

$label  CMP     $reg, #0
        B$cc    $dest
        MEND
```

MACRO 指令后面的行是*宏原型语句*。该语句定义了用于调用该宏的名称 (TestAndBranch)。它还定义了一些*参数*（$label、$dest、$reg 和 $cc）。未指定的参数将被替换为一个空字符串。 对于此宏，必须为 $dest、$reg 和 $cc 赋值，以避免出现语法错误。汇编程序会将您所提供的值替换到代码中。

可以按如下方式调用此宏:

```
test    TestAndBranch    NonZero, r0, NE
          ...
          ...
NonZero
```

在替换后将变成:

```
test    CMP     r0, #0
        BNE     NonZero
          ...
          ...
NonZero
```

2.8.2 无符号整数除法宏示例

第2-47 页的示例 2-10 显示了一个执行无符号整数除法的宏。该宏采用以下四个参数:

$Bot 存放除数的寄存器。

$Top 在执行指令之前存放被除数的寄存器。在执行指令后，该寄存器将存放余数。

$Div 存放除法的商的寄存器。如果仅需要求余数，则该参数可以为 NULL ("")。

$Temp 在计算期间使用的临时寄存器。

示例 2-10

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot            ; Produce an error message if the
        ASSERT  $Top <> $Temp           ; registers supplied are
        ASSERT  $Bot <> $Temp           ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top        ; These three only matter if $Div
            ASSERT  $Div <> $Bot        ; is not null ("")
            ASSERT  $Div <> $Temp       ;
        ENDIF
$Lab
        MOV     $Temp, $Bot             ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1     ; double it until
90      MOVLS   $Temp, $Temp, LSL #1    ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                    ; The b means search backwards
        IF      "$Div" <> ""            ; Omit next instruction if $Div is null
            MOV     $Div, #0            ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp             ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp        ; If we can, do so
        IF      "$Div" <> ""            ; Omit next instruction if $Div is null
            ADC     $Div, $Div, $Div    ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1    ; Halve $Temp,
        CMP     $Temp, $Bot             ; and loop until
        BHS     %b91                    ; less than divisor
        MEND
```

该宏会检查是否任何两个参数都没有使用相同的寄存器。如果只需要计算余数，该宏还会优化所生成的代码。

如果在汇编程序源中多次使用了 `DivMod`，则为了避免多次定义标签，该宏将使用局部标签 (90, 91)。有关详细信息，请参阅第2-12 页的*局部标签*。

第2-48 页的示例 2-11 显示了按如下方式调用此宏时它所生成的代码:

```
ratio  DivMod  r0,r5,r4,r2
```

示例 2-11

```
        ASSERT  r5 <> r4                ; Produce an error if the
        ASSERT  r5 <> r2                ; registers supplied are
        ASSERT  r4 <> r2                ; not all different
```



```
        ASSERT  r0 <> r5                ; These three only matter if $Div
        ASSERT  r0 <> r4                ; is not null ("")
        ASSERT  r0 <> r2                ;
ratio
        MOV     r2, r4                  ; Put divisor in $Temp
        CMP     r2, r5, LSR #1          ; double it until
90      MOVLS   r2, r2, LSL #1          ; 2 * r2 > r5
        CMP     r2, r5, LSR #1
        BLS     %b90                    ; The b means search backwards
        MOV     r0, #0                  ; Initialize quotient
91      CMP     r5, r2                  ; Can we subtract r2?
        SUBCS   r5, r5, r2              ; If we can, do so
        ADC     r0, r0, r0              ; Double r0

        MOV     r2, r2, LSR #1          ; Halve r2,
        CMP     r2, r4                  ; and loop until
        BHS     %b91                    ; less than divisor
```

2.9 添加符号版本

ARM 链接器符合*《ARM 体系结构的基础平台》*[BPABI]，并支持 GNU 扩展符号版本控制模型。

若要为现有符号添加符号版本，必须在同一地址处定义版本符号。版本符号的格式如下:

- 对于 *name* 的非缺省版本 *ver*，为 `name@ver`。
- 对于 *name* 的缺省 *ver*，为 `name@@ver`。

版本符号必须括在一对竖线内。

例如，定义缺省版本:

```
|my_versioned_symbol@@ver2|   ; Default version
my_asm_function PROC
                ...
                BX lr
                ENDP
```

定义非缺省版本:

```
|my_versioned_symbol@ver1|    ; Non default version
my_old_asm_function     PROC
                        ...
                        BX lr
                        ENDP
```

有关 RVCT 中的符号版本控制的完整描述，请参阅*《RealView 编译工具链接器和实用程序指南》*中的第 4 章 *访问映像符号*。

2.10 使用框架指令

若要执行下列操作之一，必须使用框架指令来描述代码使用堆栈的方式：

- 利用堆栈展开来调试应用程序
- 使用平面图或调用图执行分析。

有关这些指令的详细信息，请参阅第7-40 页的*Frame 指令*。

汇编程序使用框架指令将 DWARF 调试框架信息插入到它所生成的 ELF 格式的目标文件中。调试器执行堆栈展开和分析时需要使用这些信息。有关堆栈检查限定符的详细信息，请参阅 `*install_directory*\Documentation\Specifications\...` 中的*《ARM 体系结构的过程调用标准》*规范 `aapcs.pdf`。

有以下几点需要注意：

- 框架指令不影响汇编程序所生成的代码。
- 汇编程序不按所发出的指令验证框架指令中的信息。

2.11 汇编语言变更

表2-10 显示了当前 ARM/Thumb 汇编语言与早期单独的 ARM 和 Thumb 汇编语言之间的主要区别。汇编程序支持旧的 ARM 语法。

表2-10 自早期 ARM 汇编语言以来所发生的变更

变更	旧 ARM 语法	首选语法
LDM 和 STM 的缺省寻址模式为 IA	LDMIA, STMIA	LDM、STM
可以对 ARM 和 Thumb 中的降序满堆栈操作使用 PUSH 和 POP 助记符。	STMFD sp!, {*reglist*} LDMFD sp!, {*reglist*}	PUSH {*reglist*} POP {*reglist*}
可以对 ARM 和 Thumb 中具有循环操作而没有任何其他操作的指令使用 LSL、LSR、ASR、ROR 和 RRX 指令助记符。	MOV *Rd*, *Rn*, LSL *shift* MOV *Rd*, *Rn*, LSR *shift* MOV *Rd*, *Rn*, ASR *shift* MOV *Rd*, *Rn*, ROR *shift* MOV *Rd*, *Rn*, RRX	LSL *Rd*, *Rn*, *shift* LSR *Rd*, *Rn*, *shift* ASR *Rd*, *Rn*, *shift* ROR *Rd*, *Rn*, *shift* RRX *Rd*, *Rn*
对 PC 相对的寻址使用 *label* 格式。不能在新代码中使用 *offset* 格式。	LDR *Rd*, [pc, #*offset*]	LDR *Rd*, *label*
指定了用于双字内存访问的两个寄存器。您仍然必须遵守有关可使用的寄存器组合的规则。	LDRD *Rd*, *addr_mode*	LDRD *Rd*, *Rd2*, *addr_mode*
如果使用 {*cond*}，则它始终是所有指令的最后一个元素。	ADD{*cond*}S LDR{*cond*}SB	ADDS{*cond*} LDRSB{*cond*}
在 ARM 和 Thumb-2 代码中，可以同时使用 ARM {*cond*} 条件形式和 Thumb-2 IT 指令。汇编程序会检查这两者之间的一致性，并根据当前指令集汇编适当的代码。	ADDEQ r1, r2, r3 LDRNE r1, [r2, r3]	ITEQ E ADDEQ r1, r2, r3 LDRNE r1, [r2, r3]

此外，还具有一定程度的灵活性，这是以前的汇编程序所没有的（请参阅表 2-11）。

表2-11 减少的要求限制

减少的限制	首选语法	允许的语法
如果目标寄存器与第一个操作数相同，则可以使用指令的双寄存器形式。	ADD r1, r1, r3	ADD r1, r3

可以使用 ARM/Thumb 汇编语言为 Thumb 处理器编写源代码。

如果要为 Thumb-2 预处理器编写 Thumb 代码，则只能使用该处理器可用的指令。如果尝试使用不可用的指令，则汇编程序会生成错误消息。

如果要为 Thumb-2 处理器编写 Thumb 代码，则应尽可能地使用 16 位指令来将代码大小降到最低程度。

表2-12 显示了 Thumb 汇编语言与 ARM 汇编语言之间的主要区别。仅当旧的 Thumb 语法前面带有 `CODE16` 指令，或者用 `--16` 命令行选项汇编了源文件时，汇编程序才支持旧 Thumb 语法。

表2-12 旧 Thumb 语法与当前语言之间的区别

变更	旧 Thumb 语法	新语法
`LDM` 和 `STM` 的缺省寻址模式为 `IA`	`LDMIA`、`STMIA`	`LDM`、`STM`

必须对更新标记的指令使用 `S` 后缀。为避免与 32 位 Thumb-2 指令发生冲突，这一变更是有必要的。	`ADD r1, r2, r3`
	`SUB r4, r5, #6`
	`MOV r0, #1`
	`LSR r1, r2, #1`
		`ADDS r1, r2, r3`
		`SUBS r4, r5, #6`
		`MOVS r0, #1`
		`LSRS r1, r2, #1`

ALU 指令的首选形式会指定三个寄存器，即使目标寄存器与第一个操作数相同时也是如此。	`ADD r7, r8` `SUB r1, #80`	`ADD r7, r7, r8` `SUBS r1, r1, #80`
如果 *Rd* 和 *Rn* 都是低位寄存器，则 `MOV` *Rd*, *Rn* 将反汇编为 `ADDS` *Rd*, *Rn*, `#0`。	`MOV` *r2*, *r3* `MOV` *r8*, *r9* `CPY` *r0*, *r1* `LSL` *r2*, *r3*, `#0`	`ADDS` *r2*, *r3*, `#0` `MOV` *r8*, *r9* `MOV` *r0*, *r1* `MOVS` *r2*, *r3*
`NEG` *Rd*, *Rm* 反汇编为 `RSBS` *Rd*, *Rm*, `#0`。	`NEG` *Rd*, *Rm*	`RSBS` *Rd*, *Rm*, `#0`
如果 `NOP` 指令可用，它将取代 `MOV r8, r8`。	`- NOP`	`NOP MOV r8, r8`

第 3 章
汇编程序参考

本章提供有关 ARM® 汇编程序的一般参考资料。它包含以下几节:

本章不介绍如何编写 ARM 汇编语言。相关的教程信息，请参阅第 2 章 *编写 ARM 汇编语言*。

本章也不介绍指令、命令或伪指令。请参阅各相关章节了解这些方面的参考信息。

3.1 命令语法

本节仅与 armasm 相关。内联汇编程序是 C 和 C++ 编译器的组成部分，没有自身独立的命令语法。

armasm 命令行不区分大小写，文件名和特别指定之处除外。

使用此命令调用 ARM 汇编程序：

armasm {*options*} {*inputfile*}

其中 *options* 可以是以下任何组合，用空格分隔：

--16 使用旧 Thumb 语法时，指示汇编程序将指令解释为 Thumb® 指令。这等价于源文件开头的 CODE16 指令。使用新语法时，用 --thumb 选项可以指定 Thumb 或 Thumb-2 指令。

--32 指示汇编程序将指令解释为 ARM 指令。这是缺省设置。

--apcs [*qualifiers*]

指定是否要使用 *ARM 体系结构的程序调用标准* (AAPCS)。它也指定代码节的某些属性。有关详细信息，请参阅第 3-7 页的 *AAPCS*。

--arm 是 --32 的同义词。

--bigend 指示汇编程序汇编适合于大端 ARM 的代码。缺省为 --littleend。

--brief_diagnostics

请参阅第 3-14 页的*控制诊断消息的输出*。

--littleend 指示汇编程序汇编适合于小端 ARM 的代码。

--checkreglist

指示汇编程序检查 RLIST、LDM 和 STM 寄存器列表，以确保所有寄存器按寄存器编号升序排列。如果寄存器未按顺序列出，则给出一条警告信息。

此选项不提倡使用，将在以后的版本中删除。改为使用 --diag_warning 1206（请参阅第 3-14 页的*控制诊断消息的输出*）。

--cpu *name* 设置目标 CPU。请参阅第 3-9 页的 *CPU 名称*。

--debug 指示汇编程序生成 DWARF 调试表。--debug 是 -g 的同义词。

缺省为 DWARF 3。

—— 注意 ——

使用 `--debug` 时不保留局部符号。如果要保留局部符号来辅助调试，必须指定 `--keep`。

`--depend` *`dependfile`*

指示汇编程序将源文件相关性列表保存到 *`dependfile`* 中。这些选项适合与 make 实用程序一起使用。

`--depend_format=`*`string`*

此选项将输出相关性文件的格式更改为 UNIX 风格，以便与某些 UNIX make 程序兼容。

`string` 的值可以是以下值之一：

`unix` 生成具有 UNIX 风格路径分隔符的相关性文件。

`unix_escaped`

与 `unix` 相同，但是使用反斜线转义空格。

`unix_quoted`

与 `unix` 相同，但路径名用双引号括起。

`--diag_[error | remark | warning | suppress | style]`

请参阅第3-14 页的*控制诊断消息的输出*。

`--dllexport_all`

指示汇编程序提供所有全局符号的动态可见度，除非另行指定。编译 DLL 时可使用此选项。

`--dwarf2` 与 `--debug` 一起使用，可指示汇编程序生成 DWARF 2 调试表。

`--dwarf3` 与 `--debug` 一起使用，可指示汇编程序生成 DWARF 3 调试表。指定 `--debug` 时，这是缺省设置。

`--errors` *`errorfile`*

指示汇编程序将错误消息输出到 *`errorfile`*。

`--exceptions` 请参阅第3-16 页的*控制异常表生成*。

`--exceptions_unwind`

请参阅第3-16 页的*控制异常表生成*。

`--fpmode model`

指定浮点一致性，并设置库属性和浮点优化。请参阅第3-9 页的*浮点模型*。

`--fpu name` 选择目标*浮点单元* (FPU) 体系结构。请参阅第3-10 页的*FPU 名称*。

`-g` 是 `--debug` 的同义词。

`-i{dir} [,dir]…`

向源文件添加目录必须完全限定（请参阅第7-73 页的*GET 或 INCLUDE*）。

`--keep` 指示汇编程序在目标文件的符号表中保留局部标签，以供调试器使用（请参阅第7-77 页的*KEEP*）。

`--length` 请参阅第3-12 页的*列表输出到文件中*

`--library_type=lib`

启用链接时使用的相关库选择。

其中 `lib` 可以是以下项之一：

`standardlib`	指定在链接时选择完整的 RVCT 运行时库。这是缺省设置。
`microlib`	指定在链接时选择 C 微型库 (microlib)。

——注意——

如果使用库需要更多的特定优化，则可以将此选项与编译器、汇编程序或链接器一起使用。

将此选项与链接器一起使用可覆盖所有其他 `--library_type` 选项。

有关详细信息，请参阅:

- *库指南*中第3-3 页的*使用 microlib 构建应用程序*
- *《RealView 编译工具编译器参考指南》*中第2-55 页的 *--library_type=lib*。

`--list file` 指示汇编程序将汇编程序生成的汇编语言详细列表输出到 `file` 中。有关详细信息，请参阅第3-12 页的*列表输出到文件中*。

`-m` 指示汇编程序将源文件相关性列表输出到 `stdout`。

`--maxcache n` 将最大源缓冲区大小设置为 *n* 字节。缺省为 8MB。如果该大小小于 8MB，则 `armasm` 发出警告。

`--md` 指示汇编程序将源文件相关性列表输出到 `inputfile.d`。

`--memaccess` *`attributes`*

指定目标内存系统的内存访问属性。请参阅第3-11 页的*内存访问属性*。

注意

`--memaccess` 选项不提倡使用，将在以后的版本中删除。

`--no_code_gen`

指示汇编程序在第一轮汇编后退出。不生成目标文件。

`--no_esc` 指示汇编程序忽略 C 语言风格的转义特殊字符，如 `\n` 和 `\t`。

`--no_exceptions`

请参阅第3-16 页的*控制异常表生成*。

`--no_exceptions_unwind`

请参阅第3-16 页的*控制异常表生成*。

`--no_hide_all`

在编译 SVr4 共享对象时，使您可以控制符号可见度。所有导出的定义和参考都给定动态可见度（请参阅第7-70 页的*EXPORT 或 GLOBAL*）。

`--no_regs` 指示汇编程序不要预定义寄存器名称。请参阅第3-18 页的*预定义的寄存器和协处理器名称* 中的预定义寄存器名称列表。

此选项不提倡使用，将在以后的版本中删除。改为使用 `--regnames=none`。

`--no_terse` 请参阅第3-12 页的*列表输出到文件中*

`--no_unaligned_access`

指示汇编程序在目标文件中设置一个属性，以指明不使用未对齐访问。

`--no_warn` 关闭警告消息。

`-o` *`filename`* 命名输出目标文件。如果未指定此选项，则汇编程序创建 *`inputfilename`*`.o` 格式的目标文件名。

`--predefine "directive"`

指示汇编程序预先执行 `SET` 指令中的一个。有关详细信息，请参阅第3-11 页的*预先执行 SET 指令*。

`--[no_]reduce_paths`

启用或禁用在文件路径中删除冗余路径名信息。此选项只在 Windows 系统中有效。

Windows 系统对文件路径有 260 个字符的限制。如果存在其绝对路径名长度超过了 260 个字符的相对路径名，则可以使用 `--reduce_paths` 选项，通过使用对应的 `..` 实例匹配目录并成对删除 directory/`..` 序列，来减少绝对路径名的长度。

注意

建议优先使用 `--reduce_paths` 选项使路径长度最小，避免使用冗长和深层嵌套的路径名。

有关详细信息，请参阅 *《RealView 编译工具编译器参考指南》*中第2-77 页的*--[no_]reduce_paths*。

`--regnames=none`

指示汇编程序不要预定义寄存器名称。请参阅第3-18 页的*预定义的寄存器和协处理器名称*中的预定义寄存器名称列表。

`--regnames=callstd`

根据 `--apcs` 选项所指定的您要使用的 AAPCS 变体，定义其他寄存器名称（有关详细信息，请参阅第3-7 页的*AAPCS*）。

`--regnames=all`

定义所有的 AAPCS 寄存器，无论 `--apcs` 的值如何（有关详细信息，请参阅第3-7 页的*AAPCS*）。

`--show_cmdline`

显示汇编程序是如何处理命令行的。命令以标准化方式显示，并且所有 via 文件的内容将进行扩展。

`--split_ldm` 指示汇编程序使长 `LDM` 和 `STM` 指令出错。有关详细信息，请参阅第3-12 页的*分割长 LDM 和 STM*。不提倡使用此选项。

`--thumb` 使用 ARM 语法时，指示汇编程序将指令解释为 Thumb 指令。这等价于源文件开头的 `THUMB` 指令。

`--unaligned_access`

指示汇编程序在目标文件中设置一个属性，以指明使用未对齐访问。

`--unsafe` 可使来自不同体系结构的指令无错误地进行汇编。请参阅第3-14 页的*控制诊断消息的输出*）。

`--untyped_local_labels`

当引用 Thumb 代码中的标签时，强制汇编程序不设置 Thumb 位。有关详细信息，请参阅第4-153 页的*LDR 伪指令*。

`--via` *`file`* 指示汇编程序打开 *`file`*，并将命令行参数读取到汇编程序中。有关详细信息，请参阅 *《RealView 编译工具编译器参考指南》*中的附录 A *via 文件语法*。

`--width` 请参阅第3-12 页的*列表输出到文件中*

`--xref` 请参阅第3-12 页的*列表输出到文件中*

`inputfile` 指定汇编程序的输入文件。输入文件必须是 ARM 或 Thumb 汇编语言源文件。

3.1.1 获得可用选项列表

输入以下命令可获得可用汇编程序命令行选项的汇总:

```
armasm --help
```

3.1.2 使用环境变量指定命令行选项

您可以通过设置 `RVCT31_ASMOPT` 环境变量的值来指定命令行选项。其语法与命令行语法相同。汇编程序读取 `RVCT31_ASMOPT` 的值，并将其插入到命令字符串的前面。这意味着，`RVCT31_ASMOPT` 中指定的选项可以被命令行中的参数覆盖。

3.1.3 AAPCS

《ARM 体系结构的过程调用标准》 (AAPCS) 是 *《ARM 体系结构的应用程序二进制接口 (ABI)（基本标准）》*[BSABI] 规范的组成部分。遵循 AAPCS 编写代码可以确保分别编译和汇编的模块能够协同工作。

`--apcs` 选项指定是否要使用 AAPCS。它也指定代码节的某些属性。

有关详细信息，请参阅 *ARM 体系结构的程序调用标准*规范文件 aapcs.pdf，它位于 *install_directory*Documentation\Specifications\... 下。

——注意——

AAPCS 限定符不影响汇编程序生成的代码。它们表示程序员断言 *inputfile* 中的代码符合 AAPCS 的特定变体。它们会导致在汇编程序生成的目标文件中设置属性。链接器使用这些属性来检查文件的兼容性和选择适当的库变体。

---

qualifier 的值有：

none	指定 *inputfile* 不使用 AAPCS。未设置 AAPCS 寄存器。如果使用 none，则不允许使用其他限定符。
/interwork	指定 *inputfile* 中的代码适用于 ARM/Thumb 交互操作。有关信息，请参阅 RealView 编译工具开发指南中的第 4 章 *交互操作 ARM 和 Thumb*。
/nointerwork	指定 *inputfile* 中的代码不适用于 ARM/Thumb 交互操作。这是缺省设置。
/ropi	指定 *inputfile* 的内容是只读位置无关的。
/noropi	指定 *inputfile* 的内容不是只读位置无关的。这是缺省设置。
/pic	是 /ropi 的同义词。
/nopic	是 /noropi 的同义词。
/rwpi	指定 *inputfile* 的内容是读写位置无关的。
/norwpi	指定 *inputfile* 的内容不是读写位置无关的。这是缺省设置。
/pid	是 /rwpi 的同义词。
/nopid	是 /norwpi 的同义词。
/fpic	指定 *inputfile* 的内容是只读位置无关的代码，该代码要求 FPIC 寻址。

3.1.4 浮点模型

有个选项可指定浮点模型:

`--fpmode` *`model`*

选择目标浮点模型，并设置用于在链接时选择最适合的库的属性。

注意

这不会对您编写的代码造成任何更改。

`model` 可以是以下项之一:

`ieee_full` 由 IEEE 标准保证的所有工具、操作和表示对单精度和双精度都可用。可在运行时动态选择操作模式。

`ieee_fixed` 在舍入到最接近的数且无不精确异常条件下的 IEEE 标准。

`ieee_no_fenv` 使用舍入到最接近的数且无异常的 IEEE 标准。该模式与 Java 浮点算术模型兼容。

`std` 在非正规数清零、舍入到最接近的数及无异常条件下的 IEEE 有限值。它与 C 和 C++ 兼容。这是缺省选项。

有限值根据 IEEE 标准预测。不保证在 IEEE 模型定义的所有环境中都生成 NaN 和无穷大，或它们生成时都有相同的符号。另外，不保证 0 的符号为 IEEE 模型预测的符号。

`fast` 一些值会改变优化，其中提高执行速度会牺牲准确性。它不与 IEEE 兼容，也不是标准 C。

3.1.5 CPU 名称

有个选项可指定 CPU 名称:

`--cpu` *`name`* 设置目标 CPU。如果为错误的目标 CPU 汇编某些指令，这些指令会产生错误或警告（另请参阅第3-14 页的*控制诊断消息的输出*）。

`name` 的有效值是体系结构名称（如 `4T`、`5TE` 或 `6T2`），或部件号（如 `ARM7TDMI`）。有关体系结构的信息，请参阅*《ARM 体系结构参考手册》*。缺省为 ARM7TDMI®。

有关链接时对软件库选择的影响的详细信息，请参阅*《RealView 编译工具链接器和实用程序指南》*。

获得有效 CPU 名称列表

使用以下命令调用汇编程序，可以获得有效 CPU 和体系结构名称的列表：

```
armasm --cpu list
```

3.1.6 FPU 名称

有个选项可指定 FPU 名称：

`--fpu name` 选择目标*浮点单元* (FPU) 体系结构。如果指定了此选项，它将覆盖由 `--cpu` 选项设置的任何隐式 FPU。如果为错误的目标 FPU 汇编浮点指令，这些指令会产生错误或警告。

汇编程序设置与目标文件中的 *name* 相对应的编译属性。链接器相应地确定目标文件与选择的库之间的兼容性。

name 的有效值为：

`none` 选择无浮点体系结构。这将使汇编的目标文件与任何其他目标文件兼容。

`vfpv3` 选择符合体系结构 VFPv3 的硬件浮点单元。

`vfpv2` 选择符合体系结构 VFPv2 的硬件浮点单元。

`softvfp` 选择软浮点链接。如果不指定 `--fpu` 选项，并且选定的 `--cpu` 选项不暗含特定的 FPU，则这是缺省设置。

`softvfp+vfpv2` 选择浮点库，该库具有使用 VFP 指令的软浮点链接。

这相当于使用 `--fpu vfpv2`。

`softvfp+vfpv3` 选择浮点库，该库具有使用 VFP 指令的软浮点链接。

这相当于使用 `--fpu vfpv3`。

有关链接时这些值对软件库选择的影响的全部详细信息，请参阅*《RealView 编译工具链接器和实用程序指南》*。

获得有效 FPU 名称列表

使用以下命令调用汇编程序，可以获得有效 FPU 名称的列表：

```
armasm --fpu list
```

3.1.7 内存访问属性

使用以下命令指定目标内存系统的内存访问属性:

`--memaccess` *`attributes`*

缺省值是允许字节、半字和字的对齐加载和保存。*`attributes`* 用于修改缺省值。它们可以是下面任一个值之一:

`+L41` 启用未对齐 `LDR`。

`-L22` 不允许加载半字。

`-S22` 不允许存储半字。

`-L22-S22` 不允许加载和存储半字。

---注意---

`--memaccess` 选项不提倡使用，将在以后的版本中删除。

3.1.8 预先执行 SET 指令

您可以使用以下选项指示汇编程序预先执行某个 `SET` 指令:

`--predefine "`*`directive`*`"`

必须将 *`directive`* 包含在引号中。请参阅第7-8 页的*SETA*、*SETL 和 SETS*。汇编程序在设置变量值之前，还执行相应的 `GBLL`、`GBLS` 或 `GBLA` 指令来定义变量。

变量名称区分大小写。

---注意---

您的系统的命令行界面可能要求您输入特殊字符组合（例如 `\"`），来将字符串包含在 *`directive`* 中。另外，您也可以使用 `--via` *`file`* 来包含一个 `--predefine` 自变量。命令行界面不改变来自 `--via` 文件的参数。

3.1.9 分割长 LDM 和 STM

使用以下选项可指示汇编程序使涉及大量寄存器的 LDM 和 STM 指令出错:

```
--split_ldm
```

如果传送的寄存器最大数量超过以下值，此选项将使 LDM 和 STM 指令出错:

- 对于所有 STM 以及未加载 PC 的LDM，最大数目是 5
- 对于加载 PC 的 LDM，最大数目是 4。

避免较大的多寄存器传送数目可以减少具有下列特征的 ARM 系统上的中断等待时间:

- 没有高速缓存或写缓冲区（例如，没有高速缓存的 ARM7TDMI）
- 使用零等待状态，32 位内存。

同样，还要避免较大的多寄存器传送数目:

- 始终增加代码大小。
- 它对于高速缓存系统或有写缓冲区的处理器没有重要益处。
- 对于没有零等待状态内存的系统或有慢速外围设备的系统没有益处。在这些系统中，中断等待时间是由最慢的内存或外设访问所需的周期数决定的。这一时间一般远大于由多寄存器传送所引起的等待时间。

3.1.10 列表输出到文件中

使用以下选项将输出以列表形式写入文件:

```
--list file
```

它指示汇编程序将汇编程序生成的汇编语言详细列表输出到 *file*。

如果指定 - 作为 *file*，则将列表发送到 stdout。

如果未指定 *file*，则使用 --list= 将输出发送到 *inputfile*.lst。

注意

可使用没有文件名的 --list 将输出发送到 *inputfile*.lst。不过，不提倡使用此语法，汇编程序将发出警告。此语法将在以后的版本中删除。应改用 --list=。

使用以下命令行选项可控制 `--list` 的行为:

`--no_terse` 关闭 `terse` 标记。当此选项启用时，因条件汇编而被跳过的行不出现在列表中。如果关闭了 `terse` 选项，则这些行会出现在列表中。缺省值是启用。

`--width` 设置列表页面宽度。缺省值是 79 个字符。

`--length` 设置列表页面长度。长度为零表示不分页的列表。缺省值是 66 行。

`--xref` 指示汇编程序列出宏内、外符号的交叉引用信息，包括定义位置和使用位置。缺省值是关闭。

3.1.11 工程模板选项

工程模板是包含工程信息（如某特定配置的命令行选项）的文件。这些文件存储在工程模板工作目录中。以下选项控制工程模板的使用。

`--[no_]project=[`*`filename`*`]`

启用或禁用工程模板文件的使用。

`--reinitialize_workdir`

可用于提供工程模板的工作目录。

`--workdir=`*`directory`*

可用于提供工程模板的工作目录。

有关每个选项的详细信息，请参阅:

- *《RealView 编译工具编译器参考指南》*中第2-75 页的 *--[no_]project=filename*
- *《RealView 编译工具编译器参考指南》*中第2-78 页的*--reinitialize_workdir*
- *《RealView 编译工具编译器参考指南》*中第2-96 页的*--workdir=directory*。

3.1.12 控制诊断消息的输出

有多个选项可控制诊断消息的输出:

`--brief_diagnostics`

启用或禁用使用短格式诊断输出的模式。启用时不显示原始源语句行，并且当错误消息文本太长、一行放不下时也不换行。缺省为 `--no_brief_diagnostics`。

`--diag_style {arm|ide|gnu}`

指定用于显示诊断消息的样式:

`arm` 使用 ARM 汇编程序样式显示消息。如果未指定 `--diag_style`，则这是缺省设置。

`ide` 包括发生错误的行的行号和字符计数。这些值将显示在括号中。

`gnu` 采用 GNU 样式显示消息。

选择选项 `--diag_style=ide` 可隐式地选择选项 `--brief_diagnostics`。在命令行中显式选择 `--no_brief_diagnostics` 将覆盖 `--diag_style=ide` 隐含选择的 `--brief_diagnostics`。

无论选择选项 `--diag_style=arm` 还是选项 `--diag_style=gnu`，都不会暗含任何对 `--brief_diagnostics` 的选择。

`--diag_error` *`tag`*`[,` *`tag`*`, ...]`

将具有指定标签的诊断消息的严重性设置为错误（请参阅第3-15 页的表3-1）。

`--diag_remark` *`tag`*`[,` *`tag`*`, ...]`

将具有指定标签的诊断消息的严重性设置为备注（请参阅第3-15 页的表3-1）。

`--diag_warning` *`tag`*`[,` *`tag`*`, ...]`

将具有指定标签的诊断消息的严重性设置为警告（请参阅第3-15 页的表3-1）。

`--diag_suppress` *`tag`*`[,` *`tag`*`, ...]`

禁用具有指定标签的诊断消息。

`--unsafe` 可使来自不同体系结构的指令无错误地进行汇编。它将相应的错误消息改为警告消息，同时也禁止有关运算符优先级的警告（请参阅第3-36 页的*二元运算符*）。

四个 --diag_ 选项都需要 *tag*，即要禁止的消息的编号。可以指定多个标签。例如，要禁止具有编号 1293 和 187 的警告消息，请使用以下命令：

```
armasm --diag_suppress 1293,187 ...
```

汇编程序前缀 A 可与 --diag_error、--diag_remark 和 --diag_warning 一起使用，或在禁止消息时使用，例如：

```
armasm --diag_suppress A1293,A187 ...
```

诊断消息可以直接剪切并粘贴到命令行中。使用前缀字母是可选的。但是，如果已包括前缀字母，则它必须与 armasm 标识字母匹配。如果发现其他前缀，则汇编程序将会忽略该消息编号。

表3-1 解释了在选项说明中使用的术语*严重性*的含义。

表3-1 诊断消息的严重性

严重性	说明
灾难性错误	灾难性错误指示导致汇编终止的问题。这些错误包括命令行错误、内部错误以及丢失包含文件。
错误	错误指示违反了汇编语言的语法和语义规则。继续汇编，但不生成目标代码。
警告	警告指示代码中存在异常情况，可能有问题。继续汇编，除非检测到具有“错误”严重性的问题，否则将生成目标代码。
备注	备注指示常见但不推荐的汇编语言用法。缺省情况下不发出这些诊断消息。继续汇编，除非检测到具有“错误”严重性的问题，否则将生成目标代码。

3.1.13 控制异常表生成

有四个选项可以控制异常表的生成:

`--exceptions.` 指示汇编程序为所有遇到的函数开启异常表生成。

`--no_exceptions`
: 指示汇编程序关闭异常表生成。不生成表。这是缺省设置。

`--exceptions_unwind`
: 指示汇编程序在可能的地方为函数生成*展开*表。这是缺省设置。

`--no_exceptions_unwind`
: 指示汇编程序为每个函数生成*非展开*表。

为了更好地进行控制，可使用 `FRAME UNWIND ON` 和 `FRAME UNWIND OFF` 指令，请参阅第7-52 页的*FRAME UNWIND ON* 和第7-52 页的*FRAME UNWIND OFF*。

展开表

*函数*是由 `PROC/ENDP` 或 `FUNC/ENDFUNC` 指令进行封装的代码。

异常可以通过具有*展开*表的函数进行传播。汇编程序根据调试帧信息生成展开信息。

异常不能通过具有*非展开*表的函数进行传播。在异常处理过程中，如果遇到*非展开*表，则异常处理运行时环境将终止程序。

汇编程序可以为所有函数和非函数生成*非展开*表条目。只有当函数包含足够的用于描述函数中堆栈使用情况的 `FRAME` 指令时，汇编程序才会为该函数生成*展开*表。函数必须遵循*《ARM 体系结构的异常处理 ABI》*[EHABI] 的 9.1 节*使用约束*中列出的条件。如果汇编程序不能生成*展开*表，则生成*非展开*表。

3.2 源语句行格式

ARM 汇编语言模块的源语句行的一般格式是:

```
{symbol} {instruction|directive|pseudo-instruction} {;comment}
```

源代码行的所有三部分都是可选的。

指令不能从第一列开始。即使没有前缀符号，其前面也必须有空格。

可以用全大写字母编写指令、伪指令或命令，就像在本手册中一样。另外，您也可以用全小写编写它们。但决不能用混合大小写来编写指令、伪指令或命令。

使用空行可使代码更具可读性。

`symbol` 通常是一个标签（请参阅第3-25 页的*标签*和第3-26 页的*局部标签*）。在指令和伪指令中，它始终是一个标签。在某些命令中，它是代表一个变量或常数的符号。在各种情况下，命令说明能使这一点更清晰明了。

`symbol` 必须从第一列开始，并且不能包含任何空白字符（如空格或制表符）（请参阅第3-22 页的*符号命名规则*）。

3.3 预定义的寄存器和协处理器名称

所有的寄存器和协处理器名称都区分大小写。

3.3.1 预先声明的寄存器名称

下列寄存器名称是预先声明的：

- `r0-r15` 和 `R0-R15`
- `a1-a4`（自变量、结果或暂存寄存器，r0 到 r3 的同义词）
- `v1-v8`（变量寄存器，r4 到 r11）
- `sb` 和 `SB`（静态基址，r9）
- `ip` 和 `IP`（内部程序调用暂存寄存器，r12）
- `sp` 和 `SP`（堆栈指针，r13）
- `lr` 和 `LR`（链接寄存器，r14）
- `pc` 和 `PC`（程序计数器，r15）。

3.3.2 预先声明的扩展寄存器名称

以下扩展寄存器名称是预先声明的：

- `q0-q15` 和 `Q0-Q15`（NEON™ 四字寄存器）
- `d0-d31` 和 `D0-D31`（NEON 双字寄存器，VFP 双精度寄存器）
- `s0-s31` 和 `S0-S31`（VFP 单精度寄存器）。

3.3.3 预先声明的 XScale 寄存器名称

当为 Intel XScale CPU 进行汇编时，以下寄存器名称是预先声明的：

- `acc0-acc7` 和 `ACC0-ACC7`（XScale 累加器）。

当使用无线 MMX 为 Intel XScale CPU 进行汇编时，以下寄存器名称是预先声明的：

- `wR0-wR15`、`wr0-wr15` 和 `WR0-WR15`
- `wC0-wC15`、`wc0-wc15` 和 `WC0-WC15`
- `wCID`、`wcid` 和 `WCID`
- `wCon`、`wcon` 和 `WCON`
- `wCSSF`、`wcssf` 和 `WCSSF`
- `wCASF`、`wcasf` 和 `WCASF`。

3.3.4 预先声明的协处理器名称

下列协处理器名称和协处理器寄存器名称是预先声明的：

- `p0-p15`（协处理器 0-15）
- `c0-c15`（协处理器寄存器 0-15）。

3.4 内置变量和常数

表3-2 列出了由 ARM 汇编程序定义的内置变量。

表3-2 内置变量

{ARCHITECTURE}	存放所选择的 ARM 体系结构的名称。
{AREANAME}	存放当前区域的名称。
{ARMASM_VERSION}	存放一个随 armasm 版本增加的整数。
|ads$version|	具有与 {ARMASM_VERSION} 相同的值。
{CODESIZE}	是 {CONFIG} 的同义词。
{COMMANDLINE}	存放命令行的内容。
{CONFIG}	如果汇编程序正在汇编 ARM 代码，其值是 32；如果正在汇编 Thumb 代码，则其值是 16。
{CPU}	存放所选择的 CPU 的名称。缺省为 ARM7TDMI。如果在命令行 --cpu 选项中指定了一个体系结构，则 {CPU} 存放值“Generic ARM”。
{ENDIAN}	如果汇编程序是大端模式，其值是 big；如果是小端模式，则其值是 little。
{FPIC}	如果设置了 /fpic，则其值为 True。缺省为 False。
{FPU}	存放所选择的 fpu 的名称。缺省为 SoftVFP。
{INPUTFILE}	存放当前源文件的名称。
{INTER}	如果设置了 /inter，则其值为 True。缺省为 False。
{LINENUM}	存放指示当前源文件中行号的一个整数。
{OPT}	当前设置的列表选项的值。OPT 指令可用于保存当前列表选项，强迫改变其中的值，或恢复其原始值。
{PC} 或 .	当前指令的地址。
{PCSTOREOFFSET}	是 STR pc,[...] 或者 STM Rb,{..., pc} 指令的地址与 pc 存储输出之间的偏移量。这因所指定的 CPU 或体系结构而变化。
{ROPI}	如果设置了 /ropi，则其值为 True。缺省为 False。
{RWPI}	如果设置了 /rwpi，则其值为 True。缺省为 False。
{VAR} 或 @	存储区域位置计数器的当前值。

不能使用 SETA、SETL 或 SETS 指令来设置内置变量。内置变量可用在表达式或条件中，例如:

```
IF {ARCHITECTURE} = "4T"
```

|ads$version| 必须全部是小写。其他内置变量可以是大写、小写或混合大小写。

可以使用内置变量 {ARMASM_VERSION} 来辨别 armasm 的版本。

在 ADS 之前，ARM 汇编程序没有内置变量 {ARMASM_VERSION}。如果必须编译旧开发工具的代码版本，可以测试内置变量 |ads$version|。使用与下面类似的代码:

```
IF :DEF: |ads$version|
  ; code for RVCT or ADS
ELSE
  ; code for SDT
ENDIF
```

表3-3 列出了由 ARM 汇编程序定义的内置布尔常数。

表3-3 内置布尔常数

{FALSE}	逻辑常数“假”。
{TRUE}	逻辑常数“真”。

3.5 符号

可以使用符号来表示变量、地址和数字常数。表示地址的符号也称为*标签*。请参阅:

- *符号命名规则*
- 第3-23 页的*变量*
- 第3-23 页的*数字常数*
- 第3-23 页的*汇编时的变量替换*
- 第3-25 页的*标签*
- 第3-26 页的*局部标签*。

3.5.1 符号命名规则

下列一般性规则适用于符号名称:

- 符号名在其范围内必须是唯一的。
- 可以在符号名称中使用大写字母、小写字母、数字字符或下划线字符。符号名是区分大小写的,并且符号名中的所有字符都是有效的。
- 除了在局部标签中外,不要使用数字字符作为符号名的第一个字符(请参阅第3-26 页的*局部标签*)。
- 符号不得使用与内置变量名或预定义符号名相同的名称(请参阅第3-18 页的*预定义的寄存器和协处理器名称*和第3-20 页的*内置变量和常数*)。
- 如果使用了与一个指令助记码或指令相同的名称,应使用双竖杠来定界符号名。例如:

  ```
  ||ASSERT||
  ```

  竖杠不是符号的一部分。
- 不得使用符号 `|$a|`、`|$t|`、`|$t.x|` 或 `|$d|` 作为程序标签。这些是用于标记目标文件中的 ARM、Thumb、ThumbEE 和数据的映射符号。

如果必须在符号中使用更宽的字符范围(例如在使用编译器时),应使用单竖杠来定界符号名。例如:

```
|.text|
```

竖杠不是符号的一部分。不能在竖杠内使用竖杠、分号或换行符。

3.5.2 变量

变量的值可以在汇编过程中改变。有三种类型的变量:

- 数字
- 逻辑
- 字符串

变量的类型不能改变。

数字变量的可能值范围与数字常数或数字表达式的可能值范围相同（请参阅*数字常数*和第3-29 页的*数字表达式*）。

逻辑变量的可能值是 `{TRUE}` 或 `{FALSE}`（请参阅第3-32 页的*逻辑表达式*）。

字符串变量的可能值范围与字符串表达式的值范围相同（请参阅第3-28 页的*字符串表达式*）。

使用 `GBLA`、`GBLL`、`GBLS`、`LCLA`、`LCLL` 和 `LCLS` 指令可声明表示变量的符号，并可使用 `SETA`、`SETL` 和 `SETS` 指令为其赋值。请参阅:

- 第7-5 页的*GBLA、GBLL 和 GBLS*
- 第7-7 页的*LCLA、LCLL 和 LCLS*
- 第7-8 页的*SETA、SETL 和 SETS*。

3.5.3 数字常数

数字常数是 32 位整数。您可以使用取值范围在 0 到 2^{32}-1 之间的无符号数，或是取值范围在 -2^{31} 到 2^{3}-1 之间的有符号数来设置它们。但是，汇编程序不区分 -*n* 和 2^{32}-*n*。关系运算符（如 >=）使用无符号解释。这意味着 0 > -1 为 `{FALSE}`。

使用 `EQU` 指令可定义常数（请参阅第7-69 页的*EQU*）。在定义了数字常数后，就不能改变其值。

另请参阅第3-29 页的*数字表达式*和第3-30 页的*数字文本*。

3.5.4 汇编时的变量替换

可以使用一个字符串变量表示整行汇编语言或一行中的任何部分。在要用值替换变量的位置处使用带有 `$` 前缀的变量。$ 字符指示汇编程序在检查该行的语法之前，用该字符串替换到源代码行中。

也可以替换数字和逻辑变量。在替换之前，变量的当前值被转换为十六进制字符串（或逻辑变量的 T 或 F 值）。

如果变量名后面的字符会造成符号名的混淆，可以使用点号来标记变量名的结束（请参阅第3-22 页的*符号命名规则*）。在使用变量之前，必须设置其内容。

如果需要使用不被替换的 $ 字符，则使用 $$。它将被转换为单个 $。

可以在字符串中包含带有 $ 前缀的变量。替换方式与其他位置处的相同。

在竖杠内一般不进行替换，除非双引号内的竖杠不影响替换。

示例

```
    ; straightforward substitution
        GBLS    add4ff
        ;
add4ff  SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
        $add4ff.00                    ; invoke add4ff
        ; this produces
        ADD  r4,r4,#0xFF00

    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```

3.5.5 标签

标签是表示指令或数据在内存中地址的符号。它们可以是程序相对的、寄存器相对的或绝对的。

程序相对的标签

这些标签表示 PC，加上或减去一个数字常数。使用这些标签作为跳转指令的目标地址，或者用于访问嵌入在代码节里面的小型数据项。可以使用一个指令或其中一个数据定义命令上的标签，来定义程序相对的标签。请参阅:

- 第 7-22 页的*DCB*
- 第 7-23 页的*DCD 和 DCDU*
- 第 7-25 页的*DCFD 和 DCFDU*
- 第 7-26 页的*DCFS 和 DCFSU*
- 第 7-27 页的*DCI*
- 第 7-28 页的*DCQ 和 DCQU*
- 第 7-29 页的*DCW 和 DCWU*。

寄存器相对的标签

这些标签表示命名寄存器加上一个数字常数。它们最常用于访问数据节中的数据。您可以与存储映射一起定义它们。您可以根据存储映射中定义的标签，使用 `EQU` 指令来定义寄存器相对的附加标签。请参阅:

- 第 7-19 页的*MAP*
- 第 7-21 页的*SPACE*
- 第 7-24 页的*DCDO*
- 第 7-69 页的*EQU*。

绝对地址

这些地址是数字常数。它们是 0 到 2^{32}-1 范围内的整数。它们直接寻址内存。

3.5.6 局部标签

局部标签是 0-99 范围内的数字，后面可选择性地加上一个名称。在一个区域内，同一个数字可用于表示多个局部标签。

局部标签可用于代替汇编语言模块源代码行中的 `symbol`（请参阅第3-17 页的*源语句行格式*）：

- 即，在没有指令或命令的地方自己单独使用
- 在包含指令的行中使用
- 在包含代码或数据生成指令的行中使用。

局部标签通常在可能使用相对程序标签的地方使用（请参阅第3-25 页的*标签*）。

局部标签一般用于循环和例程内的条件代码，或者用于仅限局部使用的小型子例程。它们在宏中特别有用（请参阅第7-32 页的*MACRO 和 MEND*）。

使用 `ROUT` 指令可限制局部标签的范围（请参阅第7-80 页的*ROUT*）。引用一个局部标签时，将引用相同范围内匹配的标签。如果在该范围内的两个方向上都没有匹配的标签，汇编程序将生成一条错误消息，并且汇编失败。

即使在相同的范围内，您也可以对多个局部标签使用同一数字。缺省情况下，汇编程序将局部标签引用链接到：

- 具有相同数字的最近的局部标签（如果该范围内存在这种标签）
- 具有相同数字的下一个局部标签（如果该范围内没有前一个标签）。

如有必要，可使用可选的参数来修改此搜索模式。

语法

局部标签的语法为：

n{*routname*}

对局部标签的引用语法为：

%{F|B}{A|T}*n*{*routname*}

其中：

n	是局部标签的数字。
routname	是当前范围的名称。
%	产生引用。
F	指示汇编程序仅向前搜索。
B	指示汇编程序仅向后搜索。
A	指示汇编程序搜索所有宏命令层。
T	指示汇编程序仅查找此宏命令层。

如果既未指定 F，也未指定 B，则汇编程序先向后搜索，然后向前搜索。

如果既未指定 A，也未指定 T，则汇编程序搜索从当前层到最顶层的所有宏命令，但不搜索较低层的宏命令。

如果在标签中或对一个标签的引用中指定了 *routname*，则汇编程序将其与最近的前一个 ROUT 指令的名称进行比较。如果不匹配，则汇编程序生成一条错误消息，并且汇编失败。

3.6 表达式、文本和运算符

本节包括以下小节:

- *字符串表达式*
- 第3-29 页的*字符串文本*
- 第3-29 页的*数字表达式*
- 第3-30 页的*数字文本*
- 第3-31 页的*浮点文本*
- 第3-32 页的*相对寄存器和程序相对的表达式*
- 第3-32 页的*逻辑表达式*
- 第3-32 页的*逻辑文本*
- 第3-33 页的*运算符优先级*
- 第3-35 页的*一元运算符*
- 第3-36 页的*二元运算符*。

3.6.1 字符串表达式

字符串表达式由字符串文本、字符串变量、字符串处理运算符和括号组成。请参阅:

- 第3-23 页的*变量*
- 第3-29 页的*字符串文本*
- 第3-35 页的*一元运算符*
- 第3-37 页的*字符串处理运算符*
- 第7-8 页的*SETA*、*SETL 和 SETS*。

对于不能放在字符串文本中的字符，可使用 `:CHR:` 一元运算符将其放在字符串表达式中。允许使用从 0 到 255 的任何 ASCII 字符。

字符串表达式的值长度不能超过 512 个字符。其长度可以是零。

示例

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)
            ; sets the variable improb to the value "literal"
            ; with the left-most four characters of the
            ; contents of string variable strvar2 appended
```


3.6.2 字符串文本

字符串文本由包含在双引号字符内的一系列字符组成。字符串文本的长度受输入行长度的限制（请参阅第3-17 页的*源语句行格式*）。

如要在一个字符串中包含双引号字符或美元字符，可使用两个该字符。

也可以使用 C 字符串转义序列，除非指定了 `--no_esc` 选项（请参阅第3-2 页的*命令语法*）。

示例

```
abc     SETS    "this string contains only one "" double quote"
def     SETS    "this string contains only one $$ dollar symbol"
```

3.6.3 数字表达式

数字表达式由数字常数、数字变量、普通数字文本、二元运算符和括号组成。请参阅:

- 第3-23 页的*数字常数*
- 第3-23 页的*变量*
- 第3-30 页的*数字文本*
- 第3-36 页的*二元运算符*
- 第7-8 页的*SETA*、*SETL 和 SETS*。

如果整个表达式的取值为不包含寄存器或 PC 的值，则数字表达式可以包含相对寄存器或程序相对的表达式。

数字表达式的取值为 32 位整数。您可以将其解释为取值范围在 0 到 2^{32}-1 之间的无符号数，或是取值范围在 -2^{31} 到 2^{31}-1 之间的有符号数。但是，汇编程序不区分 -n 和 2^{32}-n。关系运算符（如 >=）使用无符号解释。这意味着 0 > -1 为 `{FALSE}`。

示例

```
a   SETA    256*256          ; 256*256 is a numeric expression
    MOV     r1,#(a*22)       ; (a*22) is a numeric expression
```

3.6.4 数字文本

数字文本可以采用下面任一种形式：

decimal-digits

0x*hexadecimal-digits*

&*hexadecimal-digits*

n_base-n-digits

'*character*'

其中：

decimal-digits 是仅使用数字 0 到 9 的字符序列。

hexadecimal-digits 是仅使用数字 0 到 9 以及字母 A 到 F 或 a 到 f 的字符序列。

*n_* 是一个介于 2 到 9 之间（包括 2 和 9）的数字，后跟一个下划线字符。

base-n-digits 是仅使用数字 0 到 (*n* -1) 的字符序列

character 是除单引号之外的任何单个字符。如果需要单引号，请使用 \'。在这种情况下，数字文本的值即为该字符的数字代码。

您不能使用任何其他字符。字符序列的取值必须是 0 到 2^{32}-1 之间（DCQ 和 DCQU 指令除外，其范围是 0 到 2^{64}-1）的整数。

示例

```
a       SETA    34906
addr    DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'             ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''         ; add 39 to contents of r2, result to r3
```


3.6.5 浮点文本

浮点文本可以采用下面任一种形式:

```
{-}digitsE{-}digits
{-}{digits}.digits{E{-}digits}
0xhexdigits
&hexdigits
0f_hexdigits
0d_hexdigits
```

其中:

digits 是仅使用数字 0 到 9 的字符序列。您可以写入大写或小写的 E。这些形式符合正常的浮点记号。

hexdigits 是仅使用数字 0 到 9 和字母 A 到 F 或 a 到 f 的字符序列。这些形式符合计算机内部的数字表示形式。输入无穷大和 NaN，或是如果您想确定您正在使用的精确位模式，则可以使用这些形式。

`0x_` 和 `&` 形式允许由任何数量的十六进制数字指定浮点位模式。

`0f_` 形式要求只能由 8 个十六进制数字指定浮点位模式。

`0d_` 形式要求只能由 16 个十六进制数字指定浮点位模式。

单精度浮点值的范围是:

- 最大值 3.40282347e+38
- 最小值 1.17549435e-38。

双精度浮点值的范围是:

- 最大值 1.79769313486231571e+308
- 最小值 2.22507385850720138e-308。

示例

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFD    3.725e15
    DCFS    0x7FC00000             ; Quiet NaN
    DCFD    &FFF0000000000000      ; Minus infinity
```

3.6.6 相对寄存器和程序相对的表达式

寄存器相对的表达式的取值是命名寄存器的值加上或减去一个数值常数（请参阅第7-19 页的*MAP*）。

程序相对的表达式的取值是*程序计数器* (PC) 的值加上或减去一个数值常数。它通常是标签与数字表达式的组合。

——注意——

在程序相对的寻址中使用的值是:

- 正在执行的指令*之后*的指令的地址
- OR `0xFFFFFFFC`（这在 ARM 代码中没有区别）
- 加上或减去该数值常数。

示例

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCD     value0
        ; n-1 DCD directives
        DCD     valuen          ; data+4*n points here
        ; more DCD directives
```

3.6.7 逻辑表达式

逻辑表达式由逻辑文本（`{TRUE}` 或 `{FALSE}`）、逻辑变量、布尔运算符、关系以及括号组合而成（请参阅第3-40 页的*布尔运算符*）。

关系由变量、文本、常数或表达式与适当的关系运算符组合而成（请参阅第3-39 页的*关系运算符*）。

3.6.8 逻辑文本

逻辑文本是:

- `{TRUE}`
- `{FALSE}`。

3.6.9 运算符优先级

汇编程序中包括一个扩展的运算符集，这些运算符用在表达式中。许多运算符与 C 语言等高级语言中对应的运算符类似（请参阅第3-35 页的*一元运算符*和第3-36 页的*二元运算符*）。

在它们的求值中有着非常严格的优先顺序:

1. 圆括号中的表达式应最先求值。
2. 运算符根据优先顺序应用。
3. 邻近的一元运算符从右向左求值。
4. 优先级相同的二元运算符从左向右求值。

armasm 和 C 中的运算符优先级

汇编程序的优先顺序与 C 语言中的并不完全一致。

例如，在 `armasm` 中，`(1 + 2 :SHR: 3)` 按 `(1 + (2 :SHR: 3))` = 1 求值。在 C 语言中等效的表达式按 `((1 + 2) >> 3)` = 0 求值。

建议您使用括号使优先级更加明显。

如果您的代码包含与 C 中的解析不同的表达式，并且没有使用 `--unsafe` 选项，则 `armasm` 通常会发出警告:

```
A1466W: Operator precedence means that expression would evaluate differently in C
```

表3-4 显示了 `armasm` 中运算符的优先顺序，并与 C 中的相应顺序进行了比较（请参阅第3-34 页的表3-5）。

在这些表中:

- 优先级最高的运算符位于列表的顶端。
- 优先级最高的运算符最先求值。
- 优先级相同的运算符从左向右求值。

表3-4 armasm 中的运算符优先级

armasm 中的优先级	等效的 C 运算符
一元运算符	一元运算符
`* / :MOD:`	`* / %`
字符串处理运算符	不适用
`:SHL: :SHR: :ROR: :ROL:`	`<< >>`

表3-4 armasm 中的运算符优先级 (续)

armasm 中的优先级	等效的 C 运算符
+ - :AND: :OR: :EOR:	+ - & |
= > >= < <= /= <>	== > >= < <= !=
:LAND: :LOR: :LEOR:	&& ||

表3-5 C 中的运算符优先级

C 优先级
一元运算符
* / %
+ -（同二元运算符）
<< >>
< <= > >=
== !=
&
^
|
&&
||

3.6.10 一元运算符

一元运算符具有最高的优先级，最先求值。一元运算符位于其操作数之前。邻近的运算符从右向左求值。

表3-6 列出了返回字符串的一元运算符。

表3-6 返回字符串的一元运算符

运算符	用法	说明
:CHR:	:CHR:A	返回 A 的 ASCII 代码字符。
:LOWERCASE:	:LOWERCASE:string	返回给定字符串，并将其中所有的大写字符转换为小写字符。
:REVERSE_CC:	:REVERSE_CC:cond_code	返回 cond_code 中条件代码的取反代码，如果 cond_code 不包含有效的条件代码，则返回错误。
:STR:	:STR:A	返回与数字表达式相对应的 8 位十六进制字符串，如果是逻辑表达式则返回字符串 "T" 或 "F"。
:UPPERCASE:	:UPPERCASE:string	返回给定字符串，并将其中所有的小写字符转换为大写字符。

表3-7 列出了返回数值的一元运算符。

表3-7 返回数值或逻辑值的一元运算符

运算符	用法	说明
?	?A	由定义符号 A 的行所生成的可执行代码的字节数。
+ 和 -	+A -A	一元加。一元减。+ 和 - 可以用在数字表达式和相对程序表达式中。
:BASE:	:BASE:A	如果 A 是相对 PC 或寄存器相对的表达式，则 :BASE: 返回其寄存器组件的编号。:BASE: 在宏中非常有用。
:CC_ENCODING:	:CC_ENCODING:cond_code	返回 cond_code 中条件代码的数值，如果 cond_code 不包含有效的条件代码，则返回错误。
:DEF:	:DEF:A	如果定义了 A，则为 {TRUE}，否则为 {FALSE}。
:INDEX:	:INDEX:A	如果 A 是寄存器相对的表达式，则 :INDEX: 返回相对该基址寄存器的偏移量。:INDEX: 在宏中非常有用。
:LEN:	:LEN:A	字符串 A 的长度。

表3-7 返回数值或逻辑值的一元运算符 (续)

运算符	用法	说明
`:LNOT:`	`:LNOT:A`	A 的逻辑补码。
`:NOT:`	`:NOT:A`	A 的按位补码。[a]
`:RCONST:`	`:RCONST:Rn`	寄存器编号，0-15 对应于 r0-r15。

a. ~ 是 `:NOT:` 的别名，例如 `~A`。

3.6.11 二元运算符

二元运算符应写在执行运算的子表达式对之间。

二元运算符比一元运算符的优先级低。二元运算符在本节中按优先顺序出现。

——注意——

该优先顺序与 C 语言中的顺序并不一致，请参阅第3-33 页的*armasm 和C 中的运算符优先级*。

乘法运算符

乘法运算符在所有二元运算符中优先级最高。它们只作用于数字表达式。

表3-8 中显示了乘法运算符。

表3-8 乘法运算符

运算符	别名	用法	说明
`*`		`A*B`	乘法
`/`		`A/B`	除法
`:MOD:`	`%`	`A:MOD:B`	A 对 B 求模

字符串处理运算符

第3-37 页的表3-9 中显示了字符串处理运算符。

在限制运算符 LEFT 和 RIGHT 中：

- A 必须是字符串
- B 必须是数字表达式。

在 CC 中，A 和 B 必须同为字符串。

表3-9 字符串处理运算符

运算符	用法	说明
:CC:	A:CC:B	B 合并到 A 的末尾
:LEFT:	A:LEFT:B	A 最左边的 B 个字符
:RIGHT:	A:RIGHT:B	A 最右边的 B 个字符

移位运算符

移位运算符用在数字表达式中，将第一个操作数按第二个操作数指定的数量移位或循环移位。

表3-10 中显示了移位运算符。

表3-10 移位运算符

运算符	别名	用法	说明
`:ROL:`		`A:ROL:B`	A 循环左移 B 位
`:ROR:`		`A:ROR:B`	A 向右循环移 B 位
`:SHL:`	`<<`	`A:SHL:B`	A 左移 B 位
`:SHR:`	`>>`	`A:SHR:B`	A 右移 B 位

——注意——

`SHR` 是逻辑移位，不影响符号位。

加法、减法和逻辑运算符

加法和减法运算符作用于数字表达式。

逻辑运算符作用于数字表达式。运算*按位*执行，即独立作用于操作数的每一位来产生结果。

表3-11 显示了加法、减法和逻辑运算符。

表3-11 加法、减法和逻辑运算符

运算符	别名	用法	说明
`+`		`A+B`	A 和 B 相加
`-`		`A-B`	A 减去 B
`:AND:`	`&&`	`A:AND:B`	A 和 B 的按位“与”运算
`:EOR:`	`^`	`A:EOR:B`	A 和 B 的按位“异或”运算
`:OR:`	`||`	`A:OR:B`	A 和 B 的按位“或”运算

关系运算符

表3-12 中显示了关系运算符。这些运算符作用于两个相同类型的操作数，产生逻辑值。

操作数可以是:

- 数字
- 程序相对的
- 寄存器相对的
- 字符串。

字符串按照 ASCII 顺序排序。如果 `A` 是字符串 `B` 的一个前导子串，或者对于两个字符串中最左边第一个不相同的字符，字符串 `A` 中的字符要小于字符串 `B` 中的字符，则它小于字符串 `B`。

算术值是无符号的，所以 `0>-1` 的值是 `{FALSE}`。

表3-12 关系运算符

运算符	别名	用法	说明
`=`	`==`	`A=B`	A 等于 B
`>`		`A>B`	A 大于 B
`>=`		`A>=B`	A 大于或等于 B
`<`		`A<B`	A 小于 B
`<=`		`A<=B`	A 小于或等于 B
`/=`	`<> !=`	`A/=B`	A 不等于 B

布尔运算符

这些运算符的优先级最低。它们对操作数执行标准的逻辑运算。

在所有三种情况下，A 和 B 必须为取值为 {TRUE} 或 {FALSE} 的表达式。

表3-13 中显示了布尔运算符。

表3-13 布尔运算符

运算符	用法	说明
:LAND:	A:LAND:B	A 和 B 的逻辑“与”运算
:LEOR:	A:LEOR:B	A 和 B 的逻辑“异或”运算
:LOR:	A:LOR:B	A 和 B 的逻辑“或”运算

3.7 诊断消息

汇编程序可提供一系列的附加诊断消息。缺省情况下，不显示这些诊断消息。但是，您可以使用命令行选项来控制汇编程序提供的消息。有关详细信息，请参阅第3-14 页的*控制诊断消息的输出*。

本节包括以下小节:

- *互锁*
- *IT 块生成*
- 第3-42 页的*Thumb 跳转目标对齐*。

3.7.1 互锁

您可获得有关代码中可能的互锁的警告消息，这些互锁是由 `--cpu` 选项选定的处理器的管道引起的。为此，请在调用汇编程序时使用以下命令行选项:

```
armasm --diag_warning 1563
```

注意

如果 `--cpu` 选项指定了多发布处理器（如 Cortex-A8），则汇编程序警告是不可预测的。

3.7.2 IT 块生成

如果您编写:

```
AREA x,CODE
THUMB
MOVNE   r0,r1 ; (1)
NOP
IT      NE
MOVNE   r0,r1 ; (2)
END
```

则汇编程序在第一个 `MOVNE` 指令前生成 `IT` 指令。

您可以在汇编 Thumb 代码时获得有关此自动生成 IT 块的警告消息。为此，请在调用汇编程序时使用以下命令行选项:

```
armasm --diag_warning 1763
```


3.7.3 Thumb 跳转目标对齐

在某些处理器中，非字对齐的 Thumb 指令有时占用一个或多个附加周期来循环执行。这意味着，确保跳转目标是字对齐的有益处。如果 Thumb 代码中的跳转目标不是字对齐的，汇编程序会发出警告。为此，请在调用汇编程序时使用以下命令行选项:

```
armasm --diag_warning 1604
```

3.8 使用 C 预处理程序

在汇编语言源文件中，可使用 C 预处理程序命令。若要执行此操作，必须在使用 armasm 汇编文件之前，使用 C 预处理程序对其进行预处理。有关详细信息，请参阅 *RealView 编译工具编译器用户指南*。

armasm 会正确解释结果文件中的 #line 命令。它能使用 #line 命令中的信息产生错误消息和 debug_line 表。

示例 3-1 显示了您编写的用于预处理和汇编 source.s 文件的命令。在此示例中，预处理程序输出一个名为 preprocessed.s 的文件，然后 armasm 汇编 preprocessed.s。

示例 3-1 预处理汇编语言源文件

```
armcc -E source.s > preprocessed.s
armasm preprocessed.s
```

第 4 章
ARM 和 Thumb 指令

本章将向您介绍 ARM 汇编程序支持的 ARM®、Thumb（32 位和 16 位）和 ThumbEE 指令。它包含以下几节:

其中，某些指令节还包含体系结构小节。没有体系结构小节的指令是 ARM 指令集和 Thumb 指令集的所有版本中都可用的。

4.1 指令汇总

表4-1 汇总出了 ARM、Thumb、Thumb-2 和 ThumbEE 指令集中的指令。您可利用该表查找本章的后面章节中所介绍的单个指令和伪指令。

——注意——

除非另有说明，否则 ThumbEE 指令与 Thumb 指令完全相同。

表4-1 指令位置

助记符	简单说明	页码	体系结构[a]
ADC, ADD	带进位加法，加法	页4-44	所有
ADR	加载相对程序或相对寄存器地址（短范围）	页4-22	所有
ADRL 伪指令	加载相对程序或相对寄存器地址（中等范围）	页4-149	x6M
AND	逻辑“与”	页4-50	所有
ASR	算术右移	页4-66	所有
B	跳转	页4-115	所有
BFC, BFI	位域清零和插入	页4-107	T2
BIC	位清零	页4-50	所有
BKPT	断点	页4-129	5
BL	带链接的跳转	页4-115	所有
BLX	带链接的跳转，更改指令集	页4-115	T
BX	跳转，更改指令集	页4-115	T
BXJ	跳转，更改为 Jazelle	页4-115	J，x7M
CBZ，CBNZ	比较，如果为（非）零，则跳转	页4-118	T2
CDP	协处理器数据处理操作	页4-121	x6M
CDP2	协处理器数据处理操作	页4-121	5，x6M
CHKA	检查数组	页4-146	EE

表4-1 指令位置 (续)

助记符	简单说明	页码	体系结构[a]
CLREX	清除独占	页4-38	K，x6M
CLZ	计算前导零数目	页4-53	5，x6M
CMN, CMP	与负值比较，比较	页4-54	所有
CPS	更改处理器状态	页4-135	6
CPY	复制	页4-56	6
DBG	调试	页4-141	7
DMB, DSB	数据内存屏障，数据同步屏障	页4-141	7，6M
ENTERX, LEAVEX	将状态更改为 ThumbEE 或更改状态 ThumbEE	页4-145	EE
EOR	异或	页4-50	所有
HB, HBL, HBLP, HBP	处理程序跳转，跳转到指定处理程序	页4-147	EE
ISB	指令同步屏障	页4-141	7，6M
IT	条件判断	页4-68	T2
LDC	加载协处理器	页4-126	x6M
LDC2	加载协处理器	页4-126	5，x6M
LDM	加载多个寄存器	页4-26	所有
LDR	加载寄存器指令	页4-11	所有
LDR 伪指令	加载寄存器伪指令	页4-153	所有
LDREX	独占加载寄存器	页4-35	6，x6M
LDREXB, LDREXH	独占加载寄存器，半字	页4-35	K，x6M
LDREXD	独占加载寄存器，双字	页4-35	K，x7M
LSL, LSR	逻辑左移，逻辑右移	页4-66	所有
MAR	从寄存器移动到 40 位累加器	页4-143	XScale
MCR	从寄存器移动到协处理器	页4-122	x6M
MCR2	从寄存器移动到协处理器	页4-122	5，x6M

表4-1 指令位置 (续)

助记符	简单说明	页码	体系结构[a]
MCRR	从寄存器移动到协处理器	页4-122	5E，x6M
MCRR2	从寄存器移动到协处理器	页4-122	6，x6M
MIA, MIAPH, MIAxy	带内部 40 位累加的乘法	页4-91	XScale
MLA	乘加	页4-73	x6M
MLS	乘减	页4-73	T2
MOV	移动	页4-56	所有
MOVT	移动到顶部	页4-59	T2
MOV32 伪指令	移动 32 位常数到寄存器	页4-151	T2
MRA	从 40 位累加器移动到寄存器	页4-143	XScale
MRC	从协处理器移动到寄存器	页4-124	所有
MRC2	从协处理器移动到寄存器	页4-124	5，x6M
MRS	从 PSR 移动到寄存器	页4-131	所有
MSR	从寄存器移动到 PSR	页4-133	所有
MUL	乘法	页4-73	所有
MVN	取反移动	页4-56	所有
NOP	无操作	页4-139	所有
ORN	逻辑“或非”	页4-50	T2
ORR	逻辑“或”	页4-50	所有
PKHBT, PKHTB	组合半字	页4-112	6，x7M
PLD	预载数据	页4-24	5E，x6M
PLI	预载指令	页4-24	7
PUSH, POP	PUSH、POP 寄存器	页4-29	所有
QADD, QDADD, QDSUB, QSUB	饱和算法	页4-94	5E，x7M

表4-1 指令位置 (续)

助记符	简单说明	页码	体系结构[a]
QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX	并行有符号饱和算法	页4-99	6，x7M
REV, REV16, REVSH	反转字节次序	页4-64	6
RBIT	反转位	页4-64	T2
RFE	从异常中返回	页4-31	T2，x7M
ROR	向右循环寄存器	页4-66	所有
RSB	反向减法	页4-44	所有
RSC	带进位反向减法	页4-44	x6M
SBC	带进位的减法	页4-44	所有
SADD8, SADD16, SASX	并行有符号算法	页4-99	6，x7M
SBFX, UBFX	有符号、无符号位域提取	页4-108	T2
SDIV	有符号除法	页4-71	7M，7R
SEL	根据 APSR GE 标记选择字节	页4-62	6，x7M
SEV	设置事件	页4-139	K，6M
SETEND	设置内存访问的端序	页4-138	6，x7M
SHADD8, SHADD16, SHASX, SHSUB8, SHSUB16, SHSAX	并行有符号均分算法	页4-99	6，x7M
SMC	安全监控调用	页4-137	Z
SMLAD	两次有符号乘加 (32 <= 32 + 16 x 16 + 16 x 16)	页4-86	6，x7M
SMLAL	有符号乘加 (64 <= 64 +32 x 32)	页4-75	x6M
SMLALxy	有符号乘加 (64 <= 64 +16 x 16)	页4-80	5E，x7M
SMLALD	两次有符号长整数乘加 (64 <= 64 + 16 x 16 + 16 x 16)	页4-88	6，x7M

表4-1 指令位置 (续)

助记符	简单说明	页码	体系结构[a]
SMLSD	两次有符号乘减累加 (32 <= 32 + 16 x 16 - 16 x 16)	页4-86	6，x7M
SMLSLD	两次有符号长整数乘减累加 (64 <= 64 + 16 x 16 - 16 x 16)	页4-88	6，x7M
SMMUL	有符号高位字乘法 (32 <= TopWord(32 x 32))	页4-84	6，x7M
SMUAD, SMUSD	有符号双乘法，并将乘积相加或相减	页4-82	6，x7M
SMULL	有符号乘法 (64 <= 32 x 32)	页4-75	x6M
SMULxy	有符号乘法 (32 <= 16 x 16)	页4-77	5E，x7M
SMULWy	有符号乘法 (32 <= 32 x 16)	页4-79	5E，x7M
SRS	存储返回状态	页4-33	T2，x7M
SSAT	有符号饱和	页4-96	6，x6M
SSAT16	有符号饱和，并行半字	页4-104	6，x7M
SSUB8, SSUB16, SSAX	并行有符号算法	页4-99	6，x7M
STC	存储协处理器	页4-126	x6M
STC2	存储协处理器	页4-126	5，x6M
STM	存储多个寄存器	页4-26	所有
STR	存储寄存器指令	页4-11	所有
STREX	独占存储寄存器	页4-35	6，x6M
STREXB, STREXH	独占存储寄存器，字节或半字	页4-35	K，x6M
STREXD	独占存储寄存器，双字	页4-35	K，x7M
SUB	减法	页4-44	所有
SUBS pc，LR	从异常中返回，无出栈	页4-48	T2，x7M
SVC（以前为 SWI）	超级用户调用	页4-130	所有

表4-1 指令位置 (续)

助记符	简单说明	页码	体系结构[a]
SWP, SWPB	交换寄存器和内存（仅 ARM）	页4-39	所有， x7M
SXT	有符号扩展	页4-109	6
SXTA	有符号扩展，带加法	页4-109	6，x7M
TBB, TBH	表跳转字节、半字	页4-119	T2
TEQ, TST	相等测试、测试	页4-60	所有
UADD8, UADD16, UASX	并行无符号算法	页4-99	6，x7M
UDIV	无符号除法	页4-71	7M，7R
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	并行无符号均分算法	页4-99	6，x7M
UMAAL	无符号长整型乘加累加 (64 <= 32 + 32 + 32 x 32)	页4-90	6，x7M
UMLAL, UMULL	无符号乘加，乘法 (64 <= 32 x 32 + 64), (64 <= 32 x 32)	页4-75	x6M
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	并行无符号饱和算法	页4-99	6，x7M
USAD8	差值的绝对值无符号求和	页4-102	6，x7M
USADA8	差值的绝对值无符号求和再累加	页4-102	6，x7M
USAT	无符号饱和	页4-96	6，x6M
USAT16	无符号饱和，并行半字	页4-104	6，x7M
USUB8, USUB16, USAX	并行无符号算法	页4-99	6，x7M
UXT	无符号扩展	页4-109	6
UXTA	无符号扩展，带加法	页4-109	6，x7M
V*	请参阅第 5 章 *NEON 和 VFP 编程*		
WFE, WFI, YIELD	等待事件，等待中断，通知	页4-139	T2，6M

a. “体系结构”列中的条目的含义如下:

所有	这些指令可用于所有版本的 ARM 体系结构。
5	这些指令可用于 ARMv5T*、ARMv6* 和 ARMv7 体系结构。
5E	这些指令可用于 ARMv5TE、ARMv6* 和 ARMv7 体系结构。
6	这些指令可用于 ARMv6* 和 ARMv7 体系结构。
6M	这些指令可用于 ARMv6-M 和 ARMv7 体系结构。
x6M	这些指令不可用于 ARMv6-M 体系结构。
7	这些指令可用于 ARMv7 体系结构。
7M	这些指令可用于 ARMv7-M 架构。
x7M	这些指令不可用于 ARMv6-M 和 ARMv7-M 架构。
7R	这些指令可用于 ARMv7-R 架构。
EE	这些指令可用在 ARM 体系结构的 ThumbEE 变体中。
J	此指令可用于 ARMv5TEJ、ARMv6* 和 ARMv7 体系结构。
K	这些指令可用于 ARMv6K 和 ARMv7 体系结构。
T	这些指令可用于 ARMv4T、ARMv5T*、ARMv6* 和 ARMv7 体系结构。
T2	这些指令可用于 ARMv6T2 和 ARMv7 体系结构。
XScale	这些指令可用于 ARM 体系结构的 XScale 版本。
Z	此指令仅当执行安全扩展后才可用。

4.2 Thumb 中的指令宽度选择

编写 ARMv6T2 或更高版本处理器的 Thumb 代码时，要注意某些指令的编码既可为 16 位，又可为 32 位。正常情况下，如果两种编码发式均可用，汇编程序会生成 16 位编码。（有关此行为的异常，请参阅*同一指令的不同行为*。）

4.2.1 指令宽度说明符 .W 和 .N

您如果要生成 32 位编码，则可使用 `.W` 宽度说明符。即使 16 位编码可用，该指令宽度说明符也会强制汇编程序生成 32 位编码。

无论代码将会被汇编为 ARM 代码还是 Thumb（ARMv6T2 或更高版本）代码，您都可在其中使用 `.W` 说明符。`.W` 说明符不会对代码的 ARM 编译产生任何影响。

如果要将指令汇编为 16 位编码，则您可使用 `.N` 宽度说明符。在这种情况下，如果指令无法编码为 16 位，或者要将代码汇编为 ARM 代码，则汇编程序会生成错误。

使用指令宽度说明符时，必须将说明符紧随在指令助记符和条件代码（如果有）之后，例如：

```
BCS.W   label   ; forces 32-bit instruction even for a short branch

B.N     label   : faults if label out of range for 16-bit instruction
```

4.2.2 同一指令的不同行为

对于向前引用，不带 `.W` 的 `LDR`、`ADR` 和 `B` 始终生成 16 位指令，即使这会导致无法访问可利用 32 位指令访问的目标。

对于外部引用，不带 `.W` 的 `LDR` 和 `B` 始终生成 32 位指令。

4.2.3 诊断警告

您可利用诊断警告来检测是否有本应该编码为 16 位的跳转指令，结果因为您使用了 `.W`，而被编码为了 32 位：

```
--diag_warning 1607
```

缺省情况下，此警告是关闭的。

注意

对于需要重新定位的指令，该诊断不会生成警告，因为尚不知道最终地址。如果该跳转超出了 32 位指令的范围，则链接器将会向其中插入胶合代码。

4.3 内存访问指令

本节包括以下小节:

- 第4-12 页的*地址对齐*

  适用于所有内存访问指令的对齐注意事项。

- 第4-13 页的*LDR 和 STR(直接偏移量)*

  带有直接偏移量、前变址直接偏移量或后变址直接偏移量的加载和存储。

- 第4-16 页的*LDR 和 STR(寄存器偏移)*

  带有寄存器偏移量、前变址寄存器偏移量或后变址寄存器偏移量的加载和存储。

- 第4-18 页的*LDR 和 STR(用户模式)*

  加载和存储,带有用户模式特权。

- 第4-20 页的*LDR(相对 pc)*

  加载寄存器。地址为 pc 加偏移量。

- 第4-22 页的*ADR*

  加载程序相对地址或寄存器相对地址。

- 第4-24 页的*PLD 和 PLI*

  预先加载地址。

- 第4-26 页的*LDM 和 STM*

  加载和存储多个寄存器。

- 第4-29 页的*PUSH 和 POP*

  将低位寄存器和 LR(可选)推入堆栈。

  从堆栈中弹出低位寄存器和 pc(可选)。

- 第4-31 页的*RFE*

  从异常中返回。

- 第4-33 页的*SRS*

  存储返回状态。

- 第4-35 页的*LDREX 和 STREX*

  独占加载和存储寄存器。

- 第4-38 页的*CLREX*
  独占清零。
- 第4-39 页的*SWP 和 SWPB*
  在寄存器和内存之间交换数据。

——注意——

同时，还有一个 `LDR` 伪指令（请参阅第4-153 页的*LDR 伪指令*）。此伪指令将汇编为 `LDR` 指令，拟或 `MOV` 或 `MVN` 指令。

4.3.1 地址对齐

在大多数情况下，您必须确保 4 字节传输的地址要 4 字节字对齐，2 字节传输的地址要 2 字节字对齐。ARMv6T2 及更高版本处理器允许未对齐访问。在 ARMv7 及更高版本处理器中，允许进行未对齐访问（缺省设置）。

在 ARMv6 和更低版本的处理器中，如果系统拥有系统协处理器 (cp15)，则您可启用对齐检查。启用了对齐检查后，未对齐的 32 位传输会引发对齐异常。

如果所有访问均是对齐的，则您可利用 `--no_unaligned_access` 命令行选项来避免链接入任何可能含有非对齐选项的库函数。

如果系统没有系统协处理器 (cp15)，或禁用了对齐检查，则:

- 对于 `STR`，所指定的地址将舍入为 4 的倍数。
- 对于 `LDR`:
  1. 所指定的地址将舍入为 4 的倍数。
  2. 从舍入生成的地址加载四字节数据。
  3. 依据舍入所生成的地址的 [1:0] 位，将所载入的数据向右循环一个、二个或三个字节。

  对于小端内存系统，这会使寻址字节占用寄存器的最低有效字节。

  对于大端内存系统，这会使寻址字节占用:

  — 如果地址的位 [0] 是 0，将占用位 [31:24]。

  — 如果地址的位 [0] 是 1，则将占用位 [15:8]。

4.3.2 LDR 和 STR（直接偏移量）

带有直接偏移量、前变址直接偏移量或后变址直接偏移量的加载和存储。

语法

```
op{type}{cond} Rt, [Rn {, #offset}]        ; immediate offset
op{type}{cond} Rt, [Rn, #offset]!          ; pre-indexed
op{type}{cond} Rt, [Rn], #offset           ; post-indexed
opD{cond} Rt, Rt2, [Rn {, #offset}]        ; immediate offset, doubleword
opD{cond} Rt, Rt2, [Rn, #offset]!          ; pre-indexed, doubleword
opD{cond} Rt, Rt2, [Rn], #offset           ; post-indexed, doubleword
```

其中：

op 是下列项之一：

- `LDR` 加载寄存器。
- `STR` 存储寄存器。

type 是下列项之一：

- `B` 无符号字节（加载时零扩展为 32 位。）
- `SB` 有符号字节（仅 `LDR`。符号扩展为 32 位。）
- `H` 无符号半字（加载时零扩展为 32 位。）
- `SH` 有符号半字（仅 `LDR`。符号扩展为 32 位。）
- `-` 如果是字，则省略。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rt 要加载或存储的寄存器。

Rn 内存地址所基于的寄存器。

offset 是偏移量。如果省略了 *offset*，则该地址为 *Rn* 中的地址。

Rt2 为附加寄存器，在双字运算中使用，用于加载或存储。

这些选项并非在所有指令集和体系结构中均可用。有关详细信息，请参阅第4-14 页的*偏移量范围和体系结构*。

偏移量范围和体系结构

表4-2 给出了偏移量的范围以及这些指令的可用性。

表4-2 偏移量和体系结构、LDR/STR、字、半字和字节

指令	直接偏移量	前变址偏移量	后变址偏移量	体系结构
ARM，字或字节[a]	-4095 到 4095	-4095 到 4095	-4095 到 4095	所有
ARM，有符号字节，半字或有符号半字	-255 到 255	-255 到 255	-255 到 255	所有
ARM，双字	-255 到 255	-255 到 255	-255 到 255	v5TE +
32 位 Thumb，字、半字，有符号半字，字节或有符号字节 [a]	-255 到 4095	-255 到 255	-255 到 255	v6T2，v7
32 位 Thumb，双字	-1020 到 1020 [c]	-1020 到 1020 [c]	-1020 到 1020 [c]	v6T2，v7
16 位 Thumb，字[b]	0 到 124[c]	不可用	不可用	所有 T
16 位 Thumb，无符号半字 [b]	0 到 62[d]	不可用	不可用	所有 T
16 位 Thumb，无符号字节 [b]	0 到 31	不可用	不可用	所有 T
16 位 Thumb，字，Rn 为 r13[e]	0 到 1020 [c]	不可用	不可用	所有 T
16 位 ThumbEE，字 [b]	-28 到 124 [c]	不可用	不可用	T-2EE
16 位 ThumbEE，字，Rn 为 r9 [e]	0 到 252 [c]	不可用	不可用	T-2EE
16 位 ThumbEE，字，Rn 为 r10 [e]	0 到 124 [c]	不可用	不可用	T-2EE

a. 对于加载字，Rt 可为 pc。加载 pc 将引发到所加载地址的跳转。对于 ARMv4，所加载的地址的位 [1:0] 必须为 0b00。对于 ARMv5 及更高版本处理器，位 [1:0] 不能为 0b10；如果位 [0] 为 1，则在 Thumb 状态下继续执行，否则要在 ARM 状态下才能继续执行。
b. Rt 和 Rn 必须在 r0-r7 范围内。
c. 必须要能被 4 整除。
d. 必须要能被 2 整除。
e. Rt 必须在 r0-r7 范围内。

双字寄存器限制

对于 Thumb-2 指令，请一定不要将 *Rt* 或 *Rt2* 指定为 sp 或 pc。

对于 ARM 指令:

- *Rt* 必须为编号为偶数的寄存器
- *Rt* 不能为 lr
- 强烈建议您不要将 r12 用作 *Rt*
- *Rt2* 必须为 R(*t* + 1)。

示例

```
    LDR     r8,[r10]            ; loads r8 from the address in r10.

    LDRNE   r2,[r5,#960]!       ; (conditionally) loads r2 from a word
                                ; 960 bytes above the address in r5, and
                                ; increments r5 by 960.

    STR     r2,[r9,#consta-struc]   ; consta-struc is an expression evaluating
                                    ; to a constant in the range 0-4095.
```

4.3.3 LDR 和 STR（寄存器偏移）

带有寄存器偏移量、前变址寄存器偏移量或后变址寄存器偏移量的加载和存储。

语法

```
op{type}{cond} Rt, [Rn, +/-Rm {, shift}]   ; register offset
op{type}{cond} Rt, [Rn, +/-Rm {, shift}]!  ; pre-indexed
op{type}{cond} Rt, [Rn], +/-Rm {, shift}   ; post-indexed
opD{cond} Rt, Rt2, [Rn, +/-Rm {, shift}]   ; register offset, doubleword
opD{cond} Rt, Rt2, [Rn, +/-Rm {, shift}]!  ; pre-indexed, doubleword
opD{cond} Rt, Rt2, [Rn], +/-Rm {, shift}   ; post-indexed, doubleword
```

其中：

op 是下列项之一：

- `LDR` 加载寄存器。
- `STR` 存储寄存器。

type 是下列项之一：

- `B` 无符号字节（加载时零扩展为 32 位。）
- `SB` 有符号字节（仅 `LDR`。符号扩展为 32 位。）
- `H` 无符号半字（加载时零扩展为 32 位。）
- `SH` 有符号半字（仅 `LDR`。符号扩展为 32 位。）
- `-` 如果是字，则省略。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rt 是要加载或存储的寄存器。

Rn 是内存地址所基于的寄存器。

Rm 为包含要用作偏移量的值的寄存器。*Rm* 不能为 r15。Thumb 代码下，不允许使用 *-m*。

shift 是一个可选的移位。

Rt2 为附加寄存器，在双字运算中使用，用于加载或存储。

这些选项并非在所有指令集和体系结构中均可用。有关详细信息，请参阅第4-17 页的*偏移寄存器和移位选项*。

偏移寄存器和移位选项

表4-3 给出了偏移量的范围以及这些指令的可用性。

表4-3 选项和体系结构，LDR/STR（寄存器偏移）

指令	+/-Rm [a]	移位	体系结构
ARM，字或字节[b]	+/-Rm	LSL #0-31　LSR #1-32 ASR #1-32　ROR #1-31　RRX	所有
ARM，有符号字节，半字或有符号半字	+/-Rm	不可用	所有
ARM，双字	+/-Rm	不可用	v5TE +
32 位 Thumb，字，半字，有符号半字，字节或有符号字节 [a]	+Rm	LSL #0-3	v6T2，v7
32 位 Thumb，双字	+Rm	不可用	v6T2，v7
16 位 Thumb，所有[c]	+Rm	不可用	所有 T
16 位 ThumbEE，字 [b]	+Rm	LSL #2（必需）	T-2EE
16 位 ThumbEE，半字，无符号半字 [b]	+Rm	LSL #1（必需）	T-2EE
16 位 ThumbEE，字节，有符号字节 [b]	+Rm	不可用	T-2EE

a. 如果显示 +/-m，则可使用 -m、+Rm 或 Rm。如果显示 +Rm，则不能使用 -m。

b. 对于加载字，Rt 可为 pc。加载 pc 会引发到所加载地址的跳转。对于 ARMv4，所加载的地址的位 [1:0] 必须为 0b00。对于 ARMv5 及更高版本的处理器，位 [1:0] 不能为 0b10；如果位 [0] 为 1，则在 Thumb 状态下继续执行，否则要在 ARM 状态才能继续执行。

c. Rt、Rn 和 Rm 必须全部在 r0-r7 范围内。

双字寄存器限制

对于 Thumb-2 指令，请一定不要将 *Rt* 或 *Rt2* 指定为 sp 或 pc。

对于 ARM 指令:

- *Rt* 必须为编号为偶数的寄存器
- *Rt* 不能为 lr
- 强烈建议您不要将 r12 用作 *Rt*
- *Rt2* 必须为 R(*t* + 1)。

4.3.4 LDR 和 STR（用户模式）

加载和存储，字节，半字或字，带有用户模式特权

在特权模式下执行这些指令时，其访问内存时所受的限制与用户模式下是相同的。

在用户模式下，这些指令的内存访问行为与正常情况下完全一致。

语法

```
op{type}T{cond} Rt, [Rn {, #offset}]      ; immediate offset (Thumb-2 only)

op{type}T{cond} Rt, [Rn] {, #offset}      ; post-indexed (ARM only)

op{type}T{cond} Rt, [Rn], +/-Rm {, shift}  ; post-indexed (register) (ARM only)
```

其中:

op 是下列项之一:

- `LDR` 加载寄存器。
- `STR` 存储寄存器。

type 是下列项之一:

- `B` 无符号字节（加载时零扩展为 32 位。）
- `SB` 有符号字节（仅 `LDR`。符号扩展为 32 位。）
- `H` 无符号半字（加载时零扩展为 32 位。）
- `SH` 有符号半字（仅 `LDR`。符号扩展为 32 位。）
- `-` 如果是字，则省略。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rt 是要加载或存储的寄存器。

Rn 是内存地址所基于的寄存器。

offset 是偏移量。如果省略偏移量，则该地址为 *Rn* 中的地址。

Rm 是一个寄存器，包含要用作偏移量的值。*Rm* 不能为 r15。

shift 是一个可选的移位。

偏移量范围和体系结构

第4-14 页的表4-2 给出了偏移量的范围以及这些指令的可用性。

表4-4 偏移量和体系结构，LDR/STR（用户模式）

指令	直接偏移量	后变址偏移量	*+/-Rm* [a]	移位	体系结构
ARM，字或字节	不可用	-4095 到 4095	+/-Rm	LSL #0-31 LSR #1-32 ASR #1-32 ROR #1-31 RRX	所有
ARM，有符号字节，半字或有符号半字	不可用	-255 到 255	+/-Rm	不可用	所有
32 位 Thumb，字，半字、有符号半字，字节或有符号字节	0 到 255	不可用	不可用		v6T2、v7

a. 可使用 -Rm、+Rm 或 Rm。

4.3.5 LDR（相对 pc）

加载寄存器。地址为 pc 加偏移量。

——注意——

另请参阅第4-148 页的*伪指令*。

语法

```
LDR{type}{cond}{.W} Rt, label
```

```
LDRD{cond} Rt, Rt2, label        ; Doubleword
```

其中：

type 是下列项之一：

- `B` 无符号字节（加载时零扩展为 32 位。）
- `SB` 有符号字节（仅 `LDR`。符号扩展为 32 位。）
- `H` 无符号半字（加载时零扩展为 32 位。）
- `SH` 有符号半字（仅 `LDR`。符号扩展为 32 位。）
- `-` 如果是字，则省略。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

`.W` 是可选的指令宽度说明符。有关详细信息，请参阅第4-21 页的*Thumb-2 中的 LDR（相对于 pc）*。

Rt 是要加载或存储的寄存器。

Rt2 是第二个要加载或存储的寄存器。

label 是一个程序相对的表达式。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

`label` 必须在当前指令的一定范围之内。有关详细信息，请参阅第4-21 页的*偏移量范围和体系结构*。

偏移量范围和体系结构

汇编程序会计算PC 相对的偏移量。如果 `label` 超出范围，汇编程序将会生成错误。

表4-5 给出了标签和当前指令之间的可能的偏移量。

表4-5 PC 相对的偏移量

指令	偏移量范围	体系结构
ARM LDR，LDRB，LDRSB，LDRH，LDRSH [a]	+/- 4095	所有
ARM LDRD	+/- 255	v5TE +
32 位 Thumb LDR，LDRB，LDRSB，LDRH，LDRSH [a]	+/- 4095	v6T2，v7
32 位 Thumb LDRD	+/- 1020 [b]	v6T2，v7
16 位 Thumb LDR [c]	0-1020 [b]	所有 T

a. 对于加载字，Rt 可为 pc。加载 pc 会引发到所加载地址的跳转。对于 ARMv4，所加载的地址的位 [1:0] 必须为 0b00。对于 ARMv5 和更高版本的处理器，位 [1:0] 不可为 0b10；如果位 [0] 为 1，则在 Thumb 状态下继续执行，否则在 ARM 状态才能继续执行。

b. 必须是 4 的倍数。

c. Rt 必须在 r0-r7 范围内。无字、半字或双字 16 位指令。

Thumb-2 中的 LDR（相对于 pc）

您可利用 .W 宽度说明符强制 LDR 在 Thumb-2 代码中生成 32 位指令。LDR.W 始终生成 32 位指令，即使利用 16 位 LDR指令就可访问目标。

对于向前引用，不带 .W 的 LDR 始终在 Thumb 代码中生成 16 位指令，即使这会导致无法访问可利用 32 位 Thumb-2 LDR 指令访问的目标。

双字寄存器限制

对于 Thumb-2 指令，请一定不要将 *Rt* 或 *Rt2* 指定为 sp 或 pc。

对于 ARM 指令:

- *Rt* 必须为编号为偶数的寄存器
- *Rt* 不能为 lr
- 强烈建议您不要将 r12 用作 *Rt*
- *Rt2* 必须为 R(*t* + 1)。

4.3.6 ADR

ADR 可将一个立即数与 pc 值相加，并会将结果写入目标寄存器。

语法

```
ADR{cond}{.W} Rd,label
```

其中：

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

.W 是可选的指令宽度说明符。有关详细信息，请参阅第4-23 页的 *Thumb-2 中的 ADR*。

Rd 是要加载的寄存器。

label 是一个程序相对的表达式。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

label 必须在当前指令的一定范围之内。有关详细信息，请参阅第4-23 页的*偏移量范围和体系结构*。

用法

ADR 所生成的代码不依赖于位置，因为其地址是由程序或寄存器产生的。

使用 ADRL 伪指令可汇编更广范围内的有效地址（请参阅第4-149 页的*ADRL 伪指令*）。

如果 *label* 与程序相关，则其值就必须为 ADR 指令所在汇编代码区域内的地址，请参阅第7-65 页的*AREA*。

如果使用 ADR 来为 BX 或 BLX 指令生成目标，当目标中包含 Thumb 指令时，您就要自己设置该地址的 Thumb 位（位 0）。

偏移量范围和体系结构

汇编程序会计算PC 相对的偏移量。如果 `label` 超出范围，汇编程序将会生成错误。

第4-21 页的表4-5 给出了标签和当前指令之间的可能的偏移量。

表4-6 PC 相对的偏移量

指令	偏移量范围	体系结构
ARM ADR	请参阅第4-42 页的*Operand2 中的常数*	所有
32 位 Thumb ADR	+/- 4095	v6T2，v7
16 位 Thumb ADR [a]	0-1020 [b]	所有 T

a. Rd 必须位于 r0-r7 范围之内。
b. 必须是 4 的倍数。

Thumb-2 中的 ADR

您可利用 .W 宽度说明符强制 ADR 在 Thumb-2 代码中生成 32 位指令。带有 .W 的 ADR 始终生成 32 位指令，即使利用 16 位指令就可访问目标地址。

对于向前引用，不带 .W 的 ADR 始终在 Thumb 代码中生成 16 位指令，即使这会导致无法访问可利用 32 位 Thumb-2 ADD 指令生成的地址。

4.3.7 PLD 和 PLI

预载数据和预载指令。处理器可向内存系统发送信号，告诉内存系统在不久的将来可能要从某个地址加载数据或指令。

是否执行 PLD 和 PLI 是可选的。如果未实现，则它们的执行方式与 NOP 类似。

语法

PL*type*{*cond*} [*Rn* {, #*offset*}]

PL*type*{*cond*} [*Rn*, +/-*Rm* {, *shift*}]

PL*type*{*cond*} *label*

其中:

type 是 D（数据）或 I（指令）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

> **注意**
>
> 这是 ARM 中的无条件指令。*cond* 只能在 Thumb-2 代码中使用，且前面要有 IT 指令。

Rn 是内存地址所基于的寄存器。

offset 是直接偏移量。如果省略偏移量，则该地址为 *Rn* 中的地址。

Rm 是一个寄存器，包含要用作偏移量的值。*Rm* 不能为 r15。

shift 是一个可选的移位。

label 是一个程序相对的表达式。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

偏移量范围

预载前，会先将偏移量应用到 *Rn* 中的值。所得结果将用作预载的内存地址。偏移量的允许范围为:

- -4095 到 +4095（ARM 指令）
- -255 到 +4095（Thumb-2 指令）。

汇编程序会计算PC 相对的偏移量。如果 *label* 超出范围，汇编程序将会生成错误。

寄存器或寄存器移位偏移量

在 ARM 中，*Rm* 中的值既可与 *Rn* 中的值相加，也可从其中减去。但在 Thumb-2 中，*Rm* 中的值只能与 *Rn* 中的值相加。所得结果将用作预载的内存地址。

移位的允许范围为：

- `LSL` 0 到 3（Thumb-2 指令）
- 对于 ARM 指令，可为以下范围之一：
  - `LSL` 0 到 31
  - `LSR` 1 到 32
  - `ASR` 1 到 32
  - `ROR` 1 到 31
  - `RRX`

预载的地址对齐

对于预载指令，系统不执行对齐检查。

体系结构

ARM `PLD` 可用于 ARMv5TE 及更高版本。ARM `PLI` 可用于 ARMv7。

32 位 Thumb `PLD` 可用于 ARMv6T2 或更高版本。32 位 Thumb `PLI` 可用于 ARMv7。

无 16 位 Thumb `PLD` 或 `PLI` 指令。

4.3.8 LDM 和 STM

加载和存储多个寄存器。寄存器 r0 到 r15 的任何组合均可在 ARM 状态下传输，但在 Thumb 状态下，会存在一些限制。

另请参阅第4-29 页的*PUSH 和 POP*。

语法

`op{addr_mode}{cond} Rn{!}, reglist{^}`

其中:

op 是下列项之一:

`LDM`	加载多个寄存器
`STM`	存储多个寄存器。

addr_mode 为下列项之一:

`IA`	在每次传送后增大地址。这是缺省情况，可以省略。
`IB`	每次传送之前增大地址（仅 ARM）。
`DA`	每次传送后减小地址（仅 ARM）。
`DB`	每次传送之前减小地址。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rn 是*基址寄存器*，存储有用于传送初始地址的 ARM 寄存器。*Rn* 不能为 r15。

`!` 是一个可选的后缀。如果有 `!`，则最终地址将写回到 *Rn* 中。

reglist 是一个或多个要加载或存储的寄存器列表，括在大括号内。可包含寄存器范围。如果包含多个寄存器或寄存器范围，则必须用逗号隔开（请参阅第4-28 页的*示例*）。

请参阅第4-27 页的*32 位 Thumb-2 指令中的 reglist 限制*。

`^` 为一个可选后缀，仅可用于 ARM 状态。不可在用户模式或系统模式下使用。它具有下列功能:

- 如果指令为 `LDM`（或 `LDMIA`），且 *reglist* 包含 pc (r15)，则除了正常的多个寄存器传送外，还要将 `SPSR` 复制到 `CPSR`。这是为了从异常处理程序返回。请仅在异常模式下使用。
- 否则，数据将被送入或送出用户模式寄存器，而不是当前模式寄存器。

32 位 Thumb-2 指令中的 reglist 限制

在 32 位 Thumb-2 指令中:

- 不可将 SP 列入列表
- 不可将 pc 列入 `STM` 指令列表中
- 不可将 pc 和 LR 列入 `LDM` 指令列表中
- 列表中必须要有两个以上的寄存器。

如果您编写的 `STM` 或 `LDM` 指令的 reglist 中仅有一个寄存器，则汇编程序将会自动用等效的 `STR` 或 `LDR` 指令替换这些指令。在比较反汇编列表与源代码时，请注意这一点。

您可利用 `--diag_warning 1645` 汇编程序命令行选项来检查是否有命令替换发生。

16 位指令

在这些指令当中，有些指令的 16 位版本可用于 Thumb-2 代码和运行于支持 Thumb 指令的处理器上的 Thumb 代码中。

对于 16 位指令，存在下列限制:

- *`reglist`* 中的所有寄存器都必须为 Lo 寄存器
- *`Rn`* 必须为 Lo 寄存器
- *`addr_mode`* 必须省略掉（或 `IA`），这意味着在每次传送后都要增加地址
- 必须指定回写。

此外，`PUSH` 和 `POP` 指令也可写为这种格式。`PUSH` 和 `POP` 的某些格式同样也是 16 位指令。有关详细信息，请参阅第4-29 页的*PUSH 和 POP*。

——注意——

这些 16 位指令不可用于 Thumb-2EE 中。

加载到 pc

加载 pc（程序计数器）将会使指令跳转到所加载地址处。

对于 ARMv4，所加载的地址的位 [1:0] 必须为 0b00。

在 ARMv5 及更高版本中:

- 位 [1:0] 不能为 0b10
- 如果位 [0] 为 1，则将会在 Thumb 状态下继续执行

- 如果位 [0] 为 0，则将会在 ARM 状态下继续执行。

加载或存储基址寄存器，带回写

在 ARM 代码或 Thumb-1 代码中，如果 *Rn* 在 *reglist* 中，且利用后缀 ! 指定了回写 则:

- 如果指令是 STM 或 STMIA，且 *Rn* 是 *reglist* 中编号最小的寄存器，则将会存储 *Rn* 的初始值
- 否则，所加载或存储的 *Rn* 值将不可预知。

在 Thumb-2 代码中，如果 *Rn* 在 *reglist* 中，且利用后缀 ! 指定了回写，则:

- 所有 32 位指令都是不可预测的
- 16 位指令的行为与在 Thumb-1 代码中相同，但反对这种用法。

示例

```
LDM     r8,{r0,r2,r9}      ; LDMIA is a synonym for LDM
STMDB   r1!,{r3-r6,r11,r12}
```

不正确的示例

```
STM     r5!,{r5,r4,r9} ; value stored for r5 unpredictable
LDMDA   r2, {}         ; must be at least one register in list
```


4.3.9 PUSH 和 POP

将寄存器推入满降序堆栈，从满降序堆栈中弹出寄存器。

语法

PUSH{*cond*} *reglist*

POP{*cond*} *reglist*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

reglist 是一个非空的寄存器列表，括在大括号内。可包含寄存器范围。如果包含多个寄存器或寄存器范围，则必须用逗号分隔。

用法

对于基址寄存器 r13 (sp) 和经调整的写回基址寄存器中的地址，PUSH 和 POP 是 STMDB 和 LDM（或 LDMIA）的同义词。在这些情况下，PUSH 和 POP 为首选助记符。

寄存器按编号顺序存储在堆栈中，编号最小的寄存器存储在最低地址。

POP，reglist 中包含 pc

此指令会产生一个跳转，会转到从堆栈弹出的存放到了 pc 中的地址处。这种跳转通常是从子程序返回，其中 lr 是在子程序开始执行时推入堆栈中的。

在 ARMv5 及更高版本中:

- 位 [1:0] 不能为 0b10
- 如果位 [0] 为 1，则将会在 Thumb 状态下继续执行
- 如果位 [0] 为 0，则将会在 ARM 状态下继续执行。

在 ARMv4 中，所加载的地址的位 [1:0] 必须为 0b00。POP 不能用于更改状态。

16 位指令

在这些指令当中，有些指令的 16 位版本可用于 Thumb-2 代码和运行于支持 Thumb 指令的处理器上的 Thumb 代码中。

对于 16 位指令，存在下列限制:

- *reglist* 中的所有寄存器必须都为 Lo 寄存器，但 PUSH 可包含 LR，POP 可包含 pc。

示例

```
PUSH    {r0,r4-r7}
PUSH    {r2,lr}
POP     {r0,r10,pc} ; no 16-bit version available
```

4.3.10 RFE

从异常中返回。

语法

```
RFE{addr_mode}{cond} Rn{!}
```

其中:

addr_mode 为下列项之一:

- IA 每次传送后增加地址（满递减堆栈）
- IB 每次传送之前增大地址（仅 ARM）
- DA 每次传送后减小地址（仅 ARM）
- DB 每次传送之前减小地址。

如果不省略 *addr_mode*，则缺省情况下为 IA。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

注意

这是 ARM 中的无条件指令。*cond* 只能在 Thumb-2 代码中使用，且前面要有 IT 指令。

Rn 指定基址寄存器。不能将 r15 用作 *Rn*。

! 是一个可选的后缀。如果有 !，则最终地址将写回到 *Rn* 中。

用法

如果您提前利用 SRS 指令保存了返回状态，则就可使用 RFE 指令从异常中返回（请参阅第4-33 页的*SRS*）。

在 Thumb-2EE 中，如果基址寄存器中的值为零，则代码将跳转到位于 HandlerBase - 4 的 NullCheck 处理程序处执行。

操作

从包含在 *Rn* 中的地址以及下面的地址加载 pc 和 CPSR。也可选择更新 *Rn*。

注释

`RFE` 可向 pc 中写入地址。对于从异常中返回后所使用的指令集，此地址的对齐必须正确:

- 如果是返回到 ARM，写入 pc 的地址必须为字对齐。
- 如果是返回到 Thumb-2，写入 pc 的地址必须为半字对齐。
- 如果是返回到 Jazelle®，则对写入 pc 的地址没有任何地址对齐限制。

如果违反这些规则，则其后果将是不可预知的。但如果指令是用于从有效的异常输入机制返回，则对于软件没有特别注意事项。

如果地址没有字对齐，则 `RFE` 将忽略 *`Rn`* 的最低有效两位。

对于 `RFE` 所生成的单个内存字，并没有从结构上定义其访问顺序。请不要在讲究访问顺序的内存映射 I/O 位置使用此指令。

如果 *`mode`* 指定了用户模式，则普通规则将应用到 CPSR 的写入操作（有关详细信息，请参阅 *《ARM 体系结构参考手册》*）。

如果 *`mode`* 指定了监控模式，则结果将是不可预知的（请参阅第4-137 页的 *SMC*）。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本。

此 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

此指令无 16 位版本。

示例

```
RFE r13!
```

4.3.11 SRS

存储返回状态。

语法

SRS{*addr_mode*}{*cond*} {r13{!}}, #*modenum*

其中:

addr_mode 为下列项之一:

- IA 每次传送后增大地址
- IB 每次传送之前增大地址(仅 ARM)
- DA 每次传送后减小地址(仅 ARM)
- DB 每次传送之前减小地址。

如果不省略 *addr_mode*,则缺省情况下为 IA。

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。

——注意——

这是 ARM 中的无条件指令。*cond* 只能在 Thumb-2 代码中使用,且前面要有 IT 指令。

! 是一个可选的后缀。如果有 !,则最终地址将会被写回到 *modenum* 所指定的模式的 r13 中。

modenum 指定模式的编号,所指定的模式的编组 r13 将被用作基址寄存器,请参阅第2-5 页的*处理器模式*。

操作

SRS 可将当前模式的 r14 和 SPSR 分别存储在 *modenum* 所指定的模式的 r13 及其后续一个字中。同时,您也可选择对 *modenum* 所指定的模式的 r13 进行更新。这与用于访问堆栈的 STM 指令的常规用法是兼容的,请参阅第4-26 页的*LDM 和 STM*。

用法

您可利用 SRS,将异常处理程序的返回状态存储到一个自动选择的不同堆栈中。

在 Thumb-2EE 中,如果基址寄存器中的值为零,则代码将跳转到位于 HandlerBase - 4 的 NullCheck 处理程序处执行。

注释

如果地址没有字对齐，则 SRS 将忽略所指定地址的最低有效两位。

对于 SRS 所生成的单个内存字，并没有从结构上定义其访问顺序。请不要在讲究访问顺序的内存映射 I/O 位置使用此指令。

在用户模式和系统模式下，SRS 的结果是不可预知的，因为这些模式均没有 SPSR。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本。

此 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

此指令无 16 位版本。

示例

```
R13_usr  EQU     16
         SRSFD   #R13_usr
```

4.3.12 LDREX 和 STREX

独占加载和存储寄存器。

语法

```
LDREX{cond} Rt, [Rn {, #offset}]
STREX{cond} Rd, Rt, [Rn {, #offset}]
LDREXB{cond} Rt, [Rn]
STREXB{cond} Rd, Rt, [Rn]
LDREXH{cond} Rt, [Rn]
STREXH{cond} Rd, Rt, [Rn]
LDREXD{cond} Rt, Rt2, [Rn]
STREXD{cond} Rd, Rt, Rt2, [Rn]
```

其中：

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是存放返回状态的目标寄存器。

Rt 是要加载或存储的寄存器。

Rt2 为进行双字加载或存储时要用到的第二个寄存器。

Rn 是内存地址所基于的寄存器。

offset 为要应用到 *Rn* 中的值的可选偏移量。*offset* 只可用于 Thumb-2 指令中。如果省略 *offset*，则就认为偏移量为 0。

LDREX

LDREX 可从内存加载数据。

- 如果物理地址有共享 TLB 属性，那么 LDREX 会将该物理地址标记为由当前处理器独占访问，并且会清除该处理器对其他任何物理地址的独占访问标签。
- 否则，会标记：执行处理器已经标记了一个物理地址，但访问尚未完毕。

STREX

STREX 可在一定条件下向内存存储数据。条件具体如下:

- 如果物理地址没有共享 TLB 属性，执行处理器有一个已标记物理地址，但尚未访问完毕，那么将会进行存储，清除该标记，并向 *Rd* 中返回值 0。
- 如果物理地址没有共享 TLB 属性，且执行处理器也没有已标记但尚未访问完毕的物理地址，那么将不会进行存储，而会向 *Rd* 返回值 1。
- 如果物理地址有共享 TLB 属性，且已被标记为由执行处理器独占访问，则将进行存储，清除标签，并向 *Rd* 返回值 0。
- 如果物理地址有共享 TLB 属性，但却没有标记为由执行处理器独占访问，则不会进行存储，而会向 *Rd* 中返回值 1。

限制

offset 不可用于 ARM 指令中。*offset* 的值可为 0-1020 范围内的任一 4 的倍数。

r15 不可用于 *Rd*、*Rt*、*Rt2* 或 *Rn*。

对于 STREX，*Rd* 一定不能与 *Rt*、*Rt2* 或 *Rn* 为同一寄存器。对于 LDREX，*Rt* 和 *Rt2* 不可为同一个寄存器。

在 ARM 指令中:

- *Rt* 必须是一个编号为偶数的寄存器，且不能为 r14
- *Rt2* 必须为 *R(d+1)*。

用法

利用 LDREX 和 STREX 可在多处理器和共享内存系统间实现进程间通信。

出于性能方面的考虑，请将相应 LDREX 指令和 STREX 指令间的指令数控制到最少。

注意

STREX 指令中所用的地址必须要与近期执行次数最多的 LDREX 指令所用的地址相同。如果使用不同的地址，则 STREX 指令的执行结果将不可预知。

体系结构

ARM LDREX 和 STREX 可用于 ARMv6 及更高版本中。

ARM LDREXB、LDREXH、LDREXD、STREXB、STREXD 和 STREXH 可用于 ARMv6K 及更高版本中。

所有这些 32 位 Thumb 指令均可用于 ARMv6T2 和 ARMv7，但 LDREXD 和 STREXD 不可用于 ARMv7-M 架构。

这些指令均无 16 位版本。

示例

```
    MOV r1, #0x1                ; load the 慘ock takení value
try
    LDREX r0, [LockAddr]        ; load the lock value
    CMP r0, #0                  ; is the lock free?
    STREXEQ r0, r1, [LockAddr]  ; try and claim the lock
    CMPEQ r0, #0                ; did this succeed?
    BNE try                     ; no - try again
    ....                        ; yes - we have the lock
```


4.3.13 CLREX

独占清零。清除执行处理器的局部记录：有地址请求进行独占访问。

语法

```
CLREX{cond}
```

其中：

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

——注意——

这是 ARM 中的无条件指令。*cond* 只能在 Thumb-2 代码中使用，且前面要有 IT 指令。

———————

用法

利用 CLREX 指令可将关系紧密的独占访问监控器返回给其开放式访问状态。这将会删除内存的虚拟存储请求。有关同步原语支持的详细信息，请参阅*《ARM 体系结构参考手册》*。

其执行还定义了 CLREX 是否清除执行处理器的全局记录：有地址请求进行独占访问。

体系结构

此 ARM 指令可用于 ARMv6K 及更高版本。

此 32 位 Thumb-2 指令可用于 ARMv6K 及更高版本的 T2 变体中。

无 16 位 Thumb CLREX 指令。

4.3.14 SWP 和 SWPB

在寄存器和内存之间交换数据。

语法

```
SWP{B}{cond} Rt, Rt2, [Rn]
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

B 是一个可选的后缀。如果存在 B，则交换一个字节。否则，交换一个 32 位字。

Rt 是目标寄存器。

Rt2 是源寄存器。*Rt2* 可与 *Rt* 为同一个寄存器。

Rn 包含内存地址。*Rn* 必须要与 *Rt* 和 *Rt2* 为不同的寄存器。

用法

您可利用 SWP 和 SWPB 来交换信息:

- 将数据从内存加载到 *Rt*。
- 将 *Rt2* 的内容保存到内存。
- 如果 *Rt2* 与 *Rt* 是同一个寄存器，则该寄存器的内容将会与内存中的内容交换。

注释

在 ARMv6 和更高版本中，不提倡使用 SWP 和 SWPB。有关可在 ARMv6 及更高版本中实现更为复杂的信息交换功能的指令的信息，请参阅第4-35 页的*LDREX 和 STREX*。

体系结构

这些 ARM 指令可用于所有版本的 ARM 体系结构中。

无 Thumb SWP 或 SWPB 指令。

4.4 通用数据处理指令

本节包括以下小节:

- 第4-41 页的*灵活的第二操作数*
- 第4-44 页的*ADD、SUB、RSB、ADC、SBC 和 RSC*

  加法、减法和反向减法，均可带进位或不带进位。
- 第4-48 页的*SUBS pc, LR*

  从异常中返回，无出栈。
- 第4-50 页的*AND、ORR、EOR、BIC 和 ORN*

  逻辑"与"、"或"、"异或"、"或非"和位清除。
- 第4-53 页的*CLZ*

  计算前导零数目。
- 第4-54 页的*CMP 和 CMN*

  比较和与负值比较。
- 第4-56 页的*MOV 和 MVN*

  移动和取反移动。
- 第4-59 页的*MOVT*

  移动到顶部，宽字。
- 第4-60 页的*TST 和 TEQ*

  测试和相等测试。
- 第4-62 页的*SEL*

  根据 APSR GE 标记的状态，从每个操作数中选择字节。
- 第4-64 页的*REV、REV16、REVSH 和 RBIT*

  反转字节或位。
- 第4-66 页的*ASR、LSL、LSR、ROR 和 RRX*

  算术右移。
- 第4-68 页的*IT*

  条件判断。
- 第4-71 页的*SDIV 和 UDIV*

  有符号除法和无符号除法。

4.4.1 灵活的第二操作数

许多 ARM 和 Thumb-2 常规数据处理指令都有一个可灵活使用的第二操作数。该操作数在每个指令的语法说明中均显示为 `Operand2`。对于 ARM 和 Thumb-2 指令，`Operand2` 的选项有一些区别。

语法

`Operand2` 有两种可能的格式：

`#constant`

`Rm``{, `*`shift`*`}`

其中：

`constant` 是取值为常数的一个表达式。对于 ARM 和 Thumb-2，该常数的范围不尽相同。有关详细信息，请参阅第4-42 页的*Operand2 中的常数*。

`Rm` 是存放第二操作数的数据的 ARM 寄存器。该寄存器中的位结构有多种移位或循环移位方式。

`shift` 是要应用于 *`Rm`* 的可选移位。它可以是下面任一个形式：

- `ASR #`*`n`* 算术右移 *n* 位。$1 \le n \le 32$。
- `LSL #`*`n`* 逻辑左移 *n* 位。$0 \le n \le 31$。
- `LSR #`*`n`* 逻辑右移 *n* 位。$1 \le n \le 32$。
- `ROR #`*`n`* 向右循环移 *n* 位。$1 \le n \le 31$。
- `RRX` 向右循环移一位，带扩展。
- *`type Rs`* 仅在 ARM 中可用，其中：
  - *`type`* 是 `ASR`、`LSL`、`LSR` 或 `ROR`。
  - *`Rs`* 是提供移位量的 ARM 寄存器。只使用最低有效字节。

——注意——

移位操作的结果就是指令中的 `Operand2`，*`Rm`* 本身不变。

Operand2 中的常数

在 ARM 指令中，*constant* 可为将 32 位字中的 8 位值向右循环移偶数位后所生成的任一值。

在 32 位 Thumb-2 指令中，*constant* 可为:

- 将 32 位字中的 8 位值向左移动任意位后所产生的任一常数。
- 格式为 `0x00XY00XY` 的任意常数。
- 格式为 `0xXY00XY00` 的任意常数。
- 格式为 `0xXYXYXYXY` 的任意常数。

此外，对于一小部分指令，*constant* 值的范围可更广一些。有关这方面的详细信息，请参阅单个指令的说明。

将 8 位值右移 2、4 或 6 位后所生成的常数可用于 ARM 数据处理指令，但不能用于 Thumb-2 指令中。所有其他 ARM 常数可用于 Thumb-2 指令中。

ASR

如果把 *Rm* 中的值视为有符号整数的二进制补码，则算术右移 *n* 位相当于将该值除以 2^n。初始时的寄存器的位 [31] 将会被复制到从左侧起第 *n* 位。

LSR 和 LSL

如果将 *Rm* 中的值视为无符号整数，则逻辑右移 *n* 位相当于将该值除以 2^n。寄存器左侧 *n* 位将会成为 0。

如果将 *Rm* 中的值视为无符号整数，则逻辑左移 *n* 位相当于将该值扩大 2^n 倍。这可能会引发不带警报的溢出。寄存器右侧 *n* 位将会成为 0。

ROR

向右循环移 *n* 位后，寄存器右侧 n 位将会移动到寄存器左侧 *n* 位中。同时，其他所有位均将右移 *n* 位（请参阅第4-43 页的图4-1）。

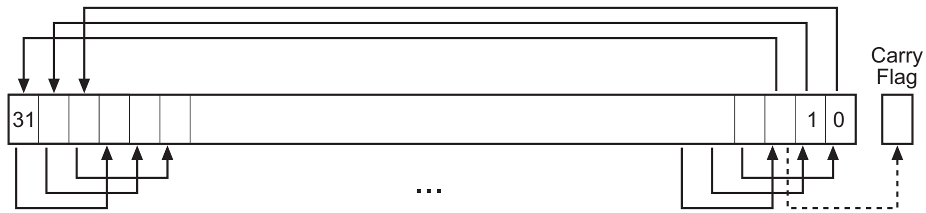


图4-1 ROR

RRX

带扩展向右循环移位将会使 *Rm* 中的值向右移动一位。进位标记将会被复制到 *Rm* 的位 [31] 中（请参阅图4-2）。

如果指定了 `S` 后缀，则 *Rm* 的位 [0] 中的值将会移位到标记中（请参阅*进位标记*）。

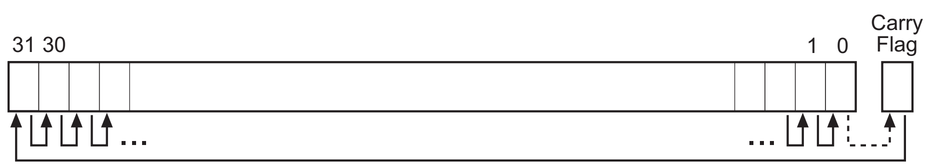


图4-2 RRX

进位标记

对于下列指令，进位标记将会更新到最后一个从 *Rm* 中移出的位中：

- `MOV`、`MVN`、`AND`、`ORR`、`ORN`, `EOR` 或 `BIC`（要使用 `S` 后缀）
- `TEQ` 或 `TST`，它们不需要 `S` 后缀。

指令替换

除了对 *constant* 进行求反或逻辑取反外，某些指令对（`ADD` 和 `SUB`、`ADC` 和 `SBC`、`AND` 和 `BIC`、`MOV` 和 `MVN`、`CMP` 和 `CMN`）是等价的。

如果无法获得 *constant* 的值，但却可对其进行求反或逻辑取反，则汇编程序将会利用其他指令进行替换，并会对 *constant* 求反或逻辑取反。

在比较反汇编列表与源代码时，请注意这一点。

您可利用 `--diag_warning 1645` 汇编程序命令行选项来检查是否有命令替换发生。

4.4.2 ADD、SUB、RSB、ADC、SBC 和 RSC

加法、减法和反向减法，均可带进位或不带进位。

另请参阅第4-99 页的*并行加法和减法*。

语法

```
op{S}{cond} {Rd}, Rn, Operand2
op{cond} {Rd}, Rn, #imm12                ; Thumb-2 only
```

其中:

op 是下列项之一:

- `ADD` 加法。
- `ADC` 带进位加法。
- `SUB` 减法。
- `RSB` 反向减法。
- `SBC` 带进位减法。
- `RSC` 带进位反向减法（仅 ARM）。

`S` 是一个可选的后缀。如果指定了 `S`，则将会更新运算结果的条件代码标记（请参阅第2-17 页的*条件执行*）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn 是存放第一个操作数的寄存器。

Operand2 是一个灵活的第二操作数。有关此选项的详细信息，请参阅第4-41 页的*灵活的第二操作数*。

imm12 可为 0-4095 范围内的任一值。只可用于 `ADD` 和 `SUB` 指令，且只可用于 Thumb-2 代码中。

用法

`ADD` 指令可将 *Rn* 中的值与 *Operand2* 中的值相加。

`SUB` 指令可从 *Rn* 中的值中减去 *Operand2* 中的值。

`RSB`（反向减法）指令可从 *Operand2* 中的值减去 *Rn* 中的值。这是很有用的，因为有了该指令，*Operand2* 的选项范围就会更大。

您可利用 `ADC`、`SBC` 和 `RSC` 来进行合成多字运算（请参阅第4-47 页的*多字算法示例*）。

`ADC`（带进位加法）指令可将 *`Rn`* 中的值和 *`Operand2`* 中的值相加，同时进位标记也会相加。

`SBC`（带进位减法）指令可从 *`Rn`* 中的值中减去 *`Operand2`* 中的值。如果清除进位标记，则结果将减 1。

`RSC`（带进位反向减法）指令可从 *`Operand2`* 中的值中减去 *`Rn`* 中的值。如果清除进位标记，则结果将减 1。

在某些情况下，汇编程序可能会进行替换指令。阅读反汇编列表时请注意这一点。有关详细信息，请参阅第4-43 页的*指令替换*。

在 Thumb-2 指令中使用 r15

在大多数 Thumb-2 指令中，请不要将 r15 用作 *`Rd`* 和任何操作数。

但也有例外，例如您可在 `ADD` 和 `SUB` 指令中，对 *`Rn`* 使用 r15，但这要求常数值 *`Operand2`* 要在 0-4095 范围内，且不能有 `S` 后缀。这些指令可用于生成 pc 相对地址。在上述情况下，pc 值的位 [1] 将会为 0，以使计算时基址始终为字对齐。

另请参阅第4-48 页的*SUBS pc, LR*。

另请参阅第4-22 页的*ADR*。

在 ARM 指令中使用 r15

如果将 r15 用作 *`Rn`*，则指令中的地址值将会加 8。

如果将 r15 用作 *`Rd`*，则:

- 代码将跳转到与结果相对应的地址执行。
- 如果使用 `S` 后缀，当前模式的 SPSR 将会被复制到 CPSR。您可利用此机制从异常中返回（请参阅 *RealView 编译工具开发指南*中的第 6 章 *处理处理器异常*）。

另请参阅第4-22 页的*ADR*。

—— 小心 ——

在用户模式和系统模式下，若是将 r15 用作了 *`Rd`*，则请不要使用 `S` 后缀。此类指令的结果是不可预测的，而且汇编程序在汇编时不会发出警告。

在所有由寄存器控制移位的数据处理指令中，都不能将 r15 用作 *Rd* 和任何操作数（请参阅第4-41 页的*灵活的第二操作数*）。

条件标记

如果指定了 S，则这些指令将会根据结果来更新 N、Z、C 和 V 标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

ADDS *Rd*, *Rn*, *#imm*	*imm* 范围为 0-7。*Rd* 和 *Rn* 必须都为 Lo 寄存器。
ADDS *Rd*, *Rn*, *Rm*	*Rd*、*Rn* 和 *Rm* 必须都为 Lo 寄存器。
ADD *Rd*, *Rd*, *Rm*	对于 ARMv6 及更早版本，*Rd* 和 *Rm* 中必须有一个为 Hi 寄存器。对于 ARMv6T2 及更高版本则没有此限制。
ADDS *Rd*, *Rd*, *#imm*	*imm* 范围为 0-255。*Rd* 必须为 Lo 寄存器。
ADCS *Rd*, *Rd*, *Rm*	*Rd*、*Rn* 和 *Rm* 必须都为 Lo 寄存器。
ADD SP, SP, *#imm*	*imm* 范围为 0-508，字对齐。
ADD *Rd*, SP, *#imm*	*imm* 范围为 0-1020，字对齐。*Rd* 必须为 Lo 寄存器。
ADD *Rd*, pc, *#imm*	*imm* 范围为 0-1020，字对齐。*Rd* 必须为 Lo 寄存器。在此指令中，pc 的位 [1:0] 将为 0。
SUBS *Rd*, *Rn*, *Rm*	*Rd*、*Rn* 和 *Rm* 必须都为 Lo 寄存器。
SUBS *Rd*, *Rn*, *#imm*	*imm* 范围为 0-7。*Rd* 和 *Rn* 都必须为 Lo 寄存器。
SUBS *Rd*, *Rd*, *#imm*	*imm* 范围为 0-255。*Rd* 必须为 Lo 寄存器。
SBCS *Rd*, *Rd*, *Rm*	*Rd*、*Rn* 和 *Rm* 必须都为 Lo 寄存器。
SUB SP, SP, *#imm*	*imm* 范围为 0-508，字对齐。
RSBS *Rd*, *Rn*, #0	*Rd* 和 *Rn* 必须为 Lo 寄存器。

示例

```
ADD     r2, r1, r3
SUBS    r8, r6, #240        ; sets the flags on the result
RSB     r4, r4, #1280       ; subtracts contents of r4 from 1280
ADCHI   r11, r0, r3         ; only executed if C flag set and Z
                            ; flag clear
RSCSLE  r0,r5,r0,LSL r4     ; conditional, flags set
```

不正确的示例

```
RSCSLE  r0,r15,r0,LSL r4    ; r15 not permitted with register
                            ; controlled shift
```

多字算法示例

下面的两个指令可将 r2 和 r3 中包含的 64 位整数与 r0 和 r1 中的包含的 64 位整数相加，并会将结果存入 r4 和 r5 中。

```
ADDS    r4, r0, r2    ; adding the least significant words
ADC     r5, r1, r3    ; adding the most significant words
```

下面的指令可完成 96 位整数的减法:

```
SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11
```

为了便于理解，对于多字值，上面的示例使用的是连续的寄存器。实际上并没有这种要求。例如，下面的示例就完全正确:

```
SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11
```

4.4.3 SUBS pc, LR

从异常中返回，无出栈。

———注意———

这是 Thumb-2 中的特殊情况指令。这一指令也可以 第4-44 页的*ADD*、*SUB*、*RSB*、*ADC*、*SBC 和 RSC* 中所描述的 `SUB` 指令的常规格式，用于 ARM 代码中。

———————

语法

`SUBS{cond} pc, LR, #imm`

其中:

imm 是一个立即数。在 Thumb-2 中，该立即数的范围为 0-255。在 ARM 代码中，它可为灵活的第二个操作数。有关详细信息，请参阅 第4-41 页的*灵活的第二操作数*。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

用法

如果堆栈中无返回状态，则您就可利用 `SUBS pc, LR` 从异常中返回。

`SUBS pc, LR` 可从链接寄存器的值中减去一个值，并将结果加载到 pc，然后再将 SPSR 复制到 CPSR。

注释

`SUBS pc, LR` 可将地址写入 pc。对于从异常中返回后所使用的指令集，此地址的对齐必须正确:

- 如果是返回到 ARM，写入 pc 的地址必须为字对齐。
- 如果是返回到 Thumb-2，写入 pc 的地址必须为半字对齐。
- 如果是返回到 Jazelle，则对写入 pc 的地址没有任何地址对齐限制。

如果违反这些规则，则其后果将是不可预知的。但如果指令是用于从有效的异常输入机制返回，则对于软件没有特别注意事项。

在 Thumb-2 中，`MOVS pc, lr` 和 `SUBS pc, lr, #0` 是同义词。

体系结构

此 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

此 ARM 指令可用于所有版本的 ARM 体系结构。

此指令无 16 位 Thumb 版本。

4.4.4 AND、ORR、EOR、BIC 和 ORN

逻辑“与”、“或”、“异或”、位清除和“或非”。

语法

op{S}{*cond*} *Rd*, *Rn*, *Operand2*

其中：

op 为下列项之一：

- AND 逻辑“与”。
- ORR 逻辑“或”。
- EOR 逻辑“异或”。
- BIC 逻辑“与非”。
- ORN 逻辑“或非”（仅 Thumb-2）。

S 是一个可选的后缀。如果指定了 S，则将会更新运算结果的条件代码标记（请参阅第2-17 页的*条件执行*）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn 是存放第一个操作数的寄存器。

Operand2 是一个灵活的第二操作数。有关此选项的详细信息，请参阅第4-41 页的*灵活的第二操作数*。

用法

AND、EOR 和 ORR 指令可对 *Rn* 和 *Operand2* 中的值按位进行“与”、“异或”和“或”运算。

BIC（位清除）指令可对 *Rn* 中的值和 *Operand2* 中值的补码按位进行逻辑“与”运算。

ORN Thumb-2 指令可对 *Rn* 中的值和 *Operand2* 中的值的补码按位进行逻辑“或”运算。

在某些情况下，汇编程序可能会用 AND 替换 BIC，用 BIC 替换 AND，用 ORR 替换 ORN，或用 ORN 替换 ORR。阅读反汇编列表时请注意这一点。有关详细信息，请参阅第4-43 页的*指令替换*。

在 Thumb-2 指令中使用 r15

在所有这些指令当中，均不能将 r15 用作 *Rd* 或任何操作数。

在 ARM 指令中使用 r15

——注意——

这些 ARM 指令中 r15 的所有用法不可用于 ARMv7。

如果将 r15 用作 *Rn*，则指令中的地址值将会加 8。

如果将 r15 用作 *Rd*，则：

- 代码将跳转到与结果相对应的地址执行。
- 如果使用 S 后缀，当前模式的 SPSR 将会被复制到 CPSR。您可利用此机制从异常中返回（请参阅 RealView 编译工具开发指南 中的第 6 章 *处理处理器异常*）。

—— 小心 ——

在用户模式和系统模式下，若是将 r15 用作了 *Rd*，则请不要使用 S 后缀。此类指令的结果是不可预测的，而且汇编程序在汇编时不会发出警告。

在所有由寄存器控制移位的数据处理指令中，都不能将 r15 用作操作数（请参阅第4-41 页的*灵活的第二操作数*）。

条件标记

如果指定了 S，则这些指令将：

- 根据结果来更新 N 和 Z 标记。
- 在 *Operand2* 的计算过程中更新 C 标记（请参阅第4-41 页的*灵活的第二操作数*）
- 不影响 V 标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令：

`ANDS` *`Rd`*, *`Rd`*, *`Rm`* *`Rd`* 和 *`Rm`* 必须都为 Lo 寄存器。

`EORS` *`Rd`*, *`Rd`*, *`Rm`* *`Rd`* 和 *`Rm`* 必须都为 Lo 寄存器。

`ORRS` *`Rd`*, *`Rd`*, *`Rm`* *`Rd`* 和 *`Rm`* 必须都为 Lo 寄存器。

`BICS` *`Rd`*, *`Rd`*, *`Rm`* *`Rd`* 和 *`Rm`* 必须都为 Lo 寄存器。

在前三种格式中，是否指定 *`OP`*`S` *`Rd`*, *`Rm`*, *`Rd`* 都可以。指令是相同的。

ARM/Thumb-2 示例

```
    AND     r9,r2,#0xFF00
    ORREQ   r2,r0,r5
    EORS    r0,r0,r3,ROR r6
    ANDS    r9, r8, #0x19
    EORS    r7, r11, #0x18181818
    BIC     r0, r1, #0xab
    ORN     r7, r11, r14, ROR #4
    ORNS    r7, r11, r14, ASR #32
```

不正确的示例

```
    EORS    r0,r15,r3,ROR r6    ; r15 not permitted with register
                                ; controlled shift
```


4.4.5 CLZ

计算前导零数目。

语法

`CLZ{`*`cond`*`} `*`Rd`*`, `*`Rm`*

其中:

cond	是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。
Rd	是目标寄存器。*Rd* 不能为 r15。
Rm	是操作数寄存器。*Rm* 不能为 r15。

用法

`CLZ` 指令可计算 *Rm* 中的值的前导零的数目，并会将结果存入 *Rd*。如果不设置源寄存器中的位，则结果为 32；如果设置位 31，则结果将为零。

条件标记

此指令不更改标记。

体系结构

此 ARM 指令可用于 ARMv5 及更高版本。

此 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

此指令无 16 位 Thumb 版本。

示例

```
CLZ     r4,r9
CLZNE   r2,r3
```

先使用 `CLZ` Thumb-2 指令，然后利用 *Rd* 中的结果值左移 *Rm*，从而标准化寄存器 *Rm* 中的值。使用 `MOVS`，而不是 `MOV`，将 *Rm* 为零的情况标记出来:

```
CLZ r5, r9
MOVS r9, r9, LSL r5
```


4.4.6 CMP 和 CMN

比较和比较取负的值。

语法

```
CMP{cond} Rn, Operand2
CMN{cond} Rn, Operand2
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rn 是存放第一个操作数的 ARM 寄存器。

Operand2 是一个灵活的第二操作数。有关此选项的详细信息，请参阅第4-41 页的*灵活的第二操作数*。

用法

这些指令可比较寄存器中的值与 *Operand2*。它们会更新结果的条件标记，但不会将结果存入任何寄存器中。

CMP 指令可从 *Rn* 中的值中减去 *Operand2* 中的值。除了结果会被丢弃以外，这与 SUBS 指令是相同的。

CMN 指令可将 *Operand2* 的值加到 *Rn* 中的值上。除了结果会被丢弃以外，这与 ADDS 指令是相同的。

在某些情况下，汇编程序可能会用 CMP 替换 CMN 或用 CMN 替换 CMP。阅读反汇编列表时请注意这一点。有关详细信息，请参阅第4-43 页的*指令替换*。

在 ARM 指令中使用 r15

——注意——

您可在这些 ARM 指令中使用 r15，但请不要在 ARMv7 中的这些指令中使用 r15。

如果将 r15 用作 *Rn*，则指令中的地址值将会加 8。

在所有由寄存器控制移位的数据处理指令中，都不能将 r15 用作操作数（请参阅第4-41 页的*灵活的第二操作数*）。

在 Thumb-2 指令中使用 r15

在所有这些指令当中，均不能将 r15 用作任何操作数。

条件标记

这些指令会根据结果更新 N、Z、C 和 V 标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

`CMP Rn, Rm`

`CMN Rn, Rm` *Rn* 和 *Rm* 必须都为 Lo 寄存器。

`CMP Rn, #imm` *Rn* 必须为 Lo 寄存器。*imm* 范围为 0-255。

示例

```
    CMP     r2, r9
    CMN     r0, #6400
    CMPGT   r13, r7, LSL #2
```

不正确的示例

```
    CMP     r2, r15, ASR r0 ; r15 not permitted with register controlled shift
```


4.4.7 MOV 和 MVN

移动和取反移动。

语法

MOV{S}{*cond*} *Rd*, *Operand2*

MOV{*cond*} *Rd*, #*imm16*

MVN{S}{*cond*} *Rd*, *Operand2*

其中:

S 是一个可选的后缀。如果指定了 S，则将会更新运算结果的条件代码标记（请参阅第2-17 页的*条件执行*）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Operand2 是一个灵活的第二操作数。有关此选项的详细信息，请参阅第4-41 页的*灵活的第二操作数*。

imm16 可为 0-65535 范围内的任一个值。

用法

MOV 指令可将 *Operand2* 中的值复制到 *Rd* 中。

MVN 指令会先获取 *Operand2* 的值，然后对该值按位进行逻辑“非”运算，最后会将结果存入 *Rd*。

在某些情况下，汇编程序可能会用 MOV 替换 MVN 或 MVN 替换 MOV。阅读反汇编列表时请注意这一点。有关详细信息，请参阅第4-43 页的*指令替换*。

在 Thumb-2 MOV 和 MVN 中使用 r15

不能将 r15 用作 *Rd*，也不可将其用在 *Operand2*、Thumb-2 MOV 或 MVN 指令中。

在 ARM MOV 和 MVN 中使用 r15

——注意——

`MOV Rd,Rm` 的语法允许将 pc 用于 Rd，或将 pc 用于 Rn，但二者不能都为 pc。 在 ARMv7 ARM 中，所有其他用法均不可行。

————

如果将 r15 用作 *Rd*，则指令中的地址值将会加 8。

如果将 r15 用作 *Rd*，则:

- 代码将跳转到与结果相对应的地址执行。
- 如果使用 S 后缀，当前模式的 SPSR 将会被复制到 CPSR。您可利用此机制从异常中返回（请参阅 RealView 编译工具开发指南 中的第 6 章 *处理处理器异常*）。

—— 小心 ——

在用户模式和系统模式下，若是将 r15 用作了 *Rd*，则请不要使用 S 后缀。此类指令的结果是不可预测的，而且汇编程序在汇编时不会发出警告。

————

在所有由寄存器控制移位的数据处理指令中，都不能将 r15 用作 *Rd* 和任何操作数（请参阅第4-41 页的*灵活的第二操作数*）。

条件标记

如果指定了 S，则这些指令将:

- 根据结果来更新 N 和 Z 标记。
- 在 *Operand2* 的计算过程中更新 C 标记（请参阅第4-41 页的*灵活的第二操作数*）
- 不影响 V 标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

`MOVS` *`Rd`*`,` *`imm`*	*`Rd`* 必须为 Lo 寄存器。*`imm`* 范围为 0-255。
`MOVS` *`Rd`*`,` *`Rm`*	*`Rd`* 和 *`Rm`* 必须都为 Lo 寄存器。
`MOV` *`Rd`*`,` *`Rm`*	在 ARMv5 及更早版本中，*`Rd`* 和 *`Rm`* 中必须至少有一个为 Hi 寄存器。在 ARMv6 及更高版本中，没有此限制。

体系结构

该 ARM 指令的 *`#imm16`* 格式可用于 ARMv6T2 中。该 ARM 指令的其他格式可用于所有版本的 ARM 体系结构中。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些 16 位 Thumb 指令可用于 ARM 体系结构的所有 T 变体中。

示例

```
    MVNNE   r11, #0xF000000B ; ARM only. This constant is not available in T2.
```

不正确的示例

```
    MVN     r15,r3,ASR r0   ; r15 not permitted with register controlled shift
```

4.4.8 MOVT

移动到顶部。将 16 位立即数写入寄存器的顶部半字中，不影响底部半字。

语法

```
MOVT{cond} Rd, #immed_16
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 为目标寄存器。*Rd* 不可为 pc。

immed_16 是一个 16 位立即数。

用法

MOVT 可将 *immed_16* 写入 *Rd*[31:16] 中。该写操作不会影响 *Rd*[15:0]。

您可利用 MOV、MOVT 指令对生成任意的 32 位常数。

另请参阅第4-151 页的*MOV32 伪指令*。

条件标记

此指令不更改标记。

体系结构

此 ARM 指令可用于 ARMv6T2 及更高版本中。

此 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

此指令无 16 位 Thumb 版本。

4.4.9 TST 和 TEQ

位测试和相等测试。

语法

TST{*cond*} *Rn*, *Operand2*

TEQ{*cond*} *Rn*, *Operand2*

其中:

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。

Rn 是存放第一个操作数的 ARM 寄存器。

Operand2 是一个灵活的第二操作数。有关此选项的详细信息,请参阅第4-41 页的*灵活的第二操作数*。

用法

这些指令可利用 *Operand2* 来测试寄存器中的值。它们会更新结果的条件标记,但不会将结果存入任何寄存器中。

TST 指令可对 *Rn* 中的值和 *Operand2* 的值按位进行逻辑"与"运算。除了结果会被丢弃以外,这与 ANDS 指令功能相同。

TEQ 指令可对 *Rn* 中的值和 *Operand2* 的值按位进行逻辑"异或"运算。除了结果会被丢弃以外,这与 EORS 指令功能相同。

利用 TEQ 指令可在不影响 V 或 C 标记的情况下,测试两个值是否相等(如同 CMP)。

TEQ 还可用于测试值的符号。比较完毕后,两个操作数的符号位逻辑"异或"运算的结果将成为 N 标记。

使用 r15

不能将 r15 用作 *Rn*,也不可将其用在 *Operand2*、Thumb-2 TST 或 TEQ 指令中。本节的剩余部分适用于 ARM 指令。

如果将 r15 用作 *Rn*,则所用的值为指令地址加 8。

在所有由寄存器控制移位的数据处理指令中,都不能将 r15 用作操作数(请参阅第4-41 页的*灵活的第二操作数*)。

条件标记

这些指令:

- 根据结果来更新 N 和 Z 标记。
- 在 `Operand2` 的计算过程中更新 C 标记（请参阅第4-41 页的*灵活的第二操作数*）
- 不影响 V 标记。

16 位指令

`TST` 指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

`TST` *`Rn`*, *`Rm`*　　*`Rn`* 和 *`Rm`* 必须都为 Lo 寄存器。

示例

```
TST     r0, #0x3F8
TEQEQ   r10, r9
TSTNE   r1, r5, ASR r1
```

不正确的示例

```
TEQ     r15, r1, ROR r0     ; r15 not permitted with register
                            ; controlled shift
```

4.4.10 SEL

根据 APSR GE 标记的状态，从每个操作数中选择字节。

语法

`SEL{cond} {Rd}, Rn, Rm`

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn 是存放第一个操作数的寄存器。

Rm 是存放第二个操作数的寄存器。

操作

SEL 指令可根据 APSR GE 标记从 *Rn* 或 *Rm* 选择字节。

- 如果设置了 GE[0]，则选取 Rn[7:0]，并存入 Rd[7:0]，否则选取 Rm[7:0]
- 如果设置了 GE[1]，则选取 Rn[15:8]，并存入 Rd[15:8]，否则选取 Rm[15:8]
- 如果设置了 GE[2]，则选取 Rn[23:16]，并存入 Rd[23:16]，否则选取 Rm[23:16]
- 如果设置了 GE[3]，则选取 Rn[31:24]，并存入 Rd[31:24]，否则选取 Rm[31:24]。

用法

不能将 r15 用作 *Rd*、*Rn* 或 *Rm*。

您可在有符号并行指令之后使用 SEL 指令，请参阅第4-99 页的*并行加法和减法*。您可利用此指令来选择多个字节或半字数据中的最大或最小值。

条件标记

此指令不更改标记。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

此指令无 16 位 Thumb 版本。

示例

```
SEL     r0, r4, r5
SELLT   r4, r0, r4
```

下面的指令序列将 `r4` 中的每个字节都设置为了与 `r1` 和 `r2` 中的相应字节相等。

```
USUB8    r4, r1, r2
SEL      r4, r2, r1
```

4.4.11 REV、REV16、REVSH 和 RBIT

在字或半字内反转字节或位的顺序。

语法

op{*cond*} *Rd*, *Rn*

其中:

op 为下列项之一:

REV	反转字中的字节顺序。
REV16	独立反转每个半字中的字节顺序。
REVSH	反转低半字中的字节顺序,并将符号扩展到 32 位。
RBIT	反转 32 位字中的位的顺序。

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。

Rd 为目标寄存器。*Rd* 不能为 r15。

Rn 为存放操作数的寄存器。*Rn* 不能为 r15。

用法

您可利用这些指令来更改端序:

REV 将 32 位大端数据变为小端数据或将 32 位小端数据变为大端数据。

REV16 将 16 位大端数据变为小端数据或将 16 位小端数据变为大端数据。

REVSH 可完成以下转换:

- 16 位带符号大端数据变为 32 位带符号小端数据
- 16 位带符号小端数据变为 32 位带符号大端数据。

条件标记

这些指令不更改标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

`REV Rd, Rm`   *Rd* 和 *Rm* 必须都为 Lo 寄存器。

`REV16 Rd, Rm`   *Rd* 和 *Rm* 必须都为 Lo 寄存器。

`REVSH Rd, Rm`   *Rd* 和 *Rm* 必须都为 Lo 寄存器。

体系结构

除 `RBIT` 外，这些 ARM 指令均可用于 ARMv6 及更高版本中。

ARM `RBIT` 指令可用于 ARMv6T2 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些 16 位 Thumb 指令可用于 ARMv6 及更高版本的所有 T 变体中。

示例

```
    REV     r3, r7
    REV16   r0, r0
    REVSH   r0, r5        ; Reverse Signed Halfword
    REVHS   r3, r7        ; Reverse with Higher or Same condition
    RBIT    r7, r8
```

4.4.12 ASR、LSL、LSR、ROR 和 RRX

算术右移、逻辑左移、逻辑右移、向右循环移和带扩展向右循环移。

这些指令与带有移位寄存器第二操作数的 MOV 功能相同。

语法

op{S}{*cond*} *Rd*, *Rm*, *Rs*

op{S}{*cond*} *Rd*, *Rm*, *#sh*

RRX{S}{*cond*} *Rd*, *Rm*

其中:

op 是 ASR、LSL、LSR 或 ROR 之一。

S 是一个可选的后缀。如果指定了 S，则将会更新运算结果的条件代码标记（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rm 是存放第一个操作数的寄存器。此操作数将会被右移。

Rs 为存放移位值的寄存器，所存放的移位值将用于移位 *Rm*。只使用最低有效字节。

sh 为一个常数移位值。该值的允许范围由具体指令决定:

ASR 允许移动 1-32 位

LSL 允许移动 0-31 位

LSR 允许移动 1-32 位

ROR 允许移动 1-31 位。

用法

ASR 可提供除以 2 后的寄存器中的有符号值。它可将符号位复制到左侧空出的位中。

LSL 可提供乘以 2 后的寄存器中的值。LSR 可提供除以 2 的 n 次幂后的寄存器中的有符号值，n 可变。这两个指令均会向空出的位中插入零。

ROR 可提供经循环移动后的寄存器中的值。从右端移出的位将会被插入到左侧空出的位中。

RRX 可提供经右移一位后的寄存器中的值。原先的进位标记将会移入位 [31]。如果有 S，则会将原先的位 [0] 存入进位标记中。

条件标记

如果指定了 S，则这些指令将会根据结果来更新 N 和 Z 标记。

如果移位值为 0，则将不会影响 C 标记。否则，C 标记将会更新为移出的最后一位。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

ASRS *Rd*, *Rm*, *#sh* *Rd* 和 *Rm* 必须都为 Lo 寄存器。

ASRS *Rd*, *Rd*, *Rs* *Rd* 和 *Rs* 必须都为 Lo 寄存器。

LSLS *Rd*, *Rm*, *#sh* *Rd* 和 *Rm* 必须都为 Lo 寄存器。

LSLS *Rd*, *Rd*, *Rs* *Rd* 和 *Rs* 必须都为 Lo 寄存器。

LSRS *Rd*, *Rm*, *#sh* *Rd* 和 *Rm* 必须都为 Lo 寄存器。

LSRS *Rd*, *Rd*, *Rs* *Rd* 和 *Rs* 必须都为 Lo 寄存器。

RORS *Rd*, *Rd*, *Rs* *Rd* 和 *Rs* 必须都为 Lo 寄存器。

示例

```
    ASR     r7, r8, r9
    LSLS    r1, r2, r3
    LSR     r4, r5, r6
    ROR     r4, r5, r6
```

4.4.13 IT

IT（条件判断）指令由下面的四个 Thumb 指令（*IT 块*）条件从句组成。这些条件可以完全相同，也可以互为逻辑反。

TBD 会将此指令移动至适当的位置。有关详细信息，请参阅 2.4 节（条件执行）。注意：该 armasm 会自动生成 IT 指令，因此代码中就不必要有显式的 IT 指令。

语法

IT{*x*{*y*{*z*}}} {*cond*}

其中：

x 是 IT 块中第二个指令的条件。

y 是 IT 块中第三个指令的条件。

z 是 IT 块中第四个指令的条件。

cond 是 IT 块中第一个指令的条件。

IT 块中第二条、第三条和第四条指令的条件是下列项之一：

T 然后。指令的执行条件为 *cond*。

E 否则。指令的执行条件为 *cond* 的反面情况。

用法

IT 块中的 16 位指令，除了 CMP、CMN 和 TST 外，都不会影响条件代码标记。如果要影响条件代码标记，则您可使用带有 AL 条件的 IT，不让 IT 块中的指令带有条件。

IT 块中的指令还必须指定语法的 {*cond*} 部分中的条件。汇编程序会根据 IT 指令中的条件来验证该条件。

汇编程序可在 ARM 汇编期间，接受 IT 指令，并会根据后续指令中的条件验证这些指令，但不会生成任何代码。

限制

不允许在 IT 块中使用下面的指令：

- IT
- 条件跳转
- CBZ 和 CBNZ
- TBB 和 TBH

- CPS、CPSID 和 CPSIE
- SETEND。

此外，IT 块的最后一条指令可为无条件跳转指令。

条件标记

此指令不更改标记。

Exceptions

IT 指令与相应的 IT 块间或 IT 块内均可能会发生异常。如果发生异常，则会跳转到异常处理程序，同时相应的返回信息会存入 r14 和 SPSR 中。

与其他情况一样，可利用异常返回指令来从异常中返回，IT 块将会继续正常执行。这是 pc 修改指令跳转到 IT 块中的指令的唯一方法。

体系结构

此 16 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

示例

```
ITTE   NE        ; assemblers can allow IT to be omitted
ANDNE  r0,r0,r1  ; 16-bit AND, not ANDS
ADDSNE r2,r2,#1  ; 32-bit ADDS (16-bit ADDS does not set flags in IT block)
MOVEQ  r2,r3     ; 16-bit MOV(CPY)
ITT    AL        ; emit 2 non-flag setting 16-bit instructions
ADDAL  r0,r0,r1  ; 16-bit ADD, not ADDS
SUBAL  r2,r2,#1  ; 16-bit SUB, not SUB
ADD    r0,r0,r1  ; expands into 32-bit ADD
IT     NE
ADD    r0,r0,r1  ; syntax error: no condition code used in IT block
```

注释

跳入 IT 块

除 *Exceptions* 中所述方法外，不管是利用跳转指令还是任何其他 pc 更改指令，均不能跳转到 IT 块中的指令。

跳出 IT 块

除最后一条指令外，不可在 IT 块中使用 pc 修改指令。

在 IT 块中跳转

IT 块中的跳转必须为无条件跳转（IT 条件会使它们变为条件跳转）。因此，在 IT 块内设置条件跳转非常有用，因为这可扩大 IT 块的控制范围：

```
ITT    EQ
MOVEQ  r0,r1
BEQ    dloop
```

IT 块中不可预知结果的指令

如果 IT 块中有不可预知结果的指令，汇编程序就会发出警告，例如 B、BL 和 CPS。同时，如果有 pc 更改指令也会发出警告，例如 BX、CZB 和 RFE。

汇编程序不会考虑 IT 块中的命令。

BKPT　　IT 块中 BKPT 指令总会得到执行，即使无法满足其条件。

4.4.14 SDIV 和 UDIV

有符号除法和无符号除法。

语法

```
SDIV{cond} {Rd}, Rn, Rm
UDIV{cond} {Rd}, Rn, Rm
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn 为存放被除数的寄存器。

Rm 为存放除数的寄存器。

寄存器限制

pc 或 sp 不可用于 `Rd`、`Rn` 或 `Rm`。

体系结构

这些 32 位 Thumb 指令仅可用于 ARMv7-R 和 ARMv7-M。

无 ARM 或 16 位 Thumb `SDIV` 和 `UDIV` 指令。

4.5 乘法指令

本节包括以下小节:

- 第4-73 页的*MUL、MLA 和 MLS*

  乘法、乘加和乘减（32 位乘 32 位，结果取低 32 位）。

- 第4-75 页的*UMULL、UMLAL、SMULL 和 SMLAL*

  无符号长乘法、乘加和有符号长乘法、乘加（32 位乘 32 位，结果或累加器均为 64 位）。

- 第4-77 页的*SMULxy 和 SMLAxy*

  有符号乘法和有符号乘加（16 位乘 16 位，结果为 32 位）。

- 第4-79 页的*SMULWy 和 SMLAWy*

  有符号乘法和有符号乘加（32 位乘 16 位，结果取高 32 位）。

- 第4-80 页的*SMLALxy*

  有符号乘加（16 位乘 16 位，64 位累加器）。

- 第4-82 页的*SMUAD{X} 和 SMUSD{X}*

  两次 16 位有符号乘法运算，然后将乘积相加或相减。

- 第4-84 页的*SMMUL、SMMLA 和 SMMLS*

  乘法、乘加和乘减（32 位乘 32 位，结果取高 32 位）。

- 第4-86 页的*SMLAD 和 SMLSD*

  两次 16 位有符号乘法，然后将 32 位乘积的和或差进行 32 位累加。

- 第4-88 页的*SMLALD 和 SMLSLD*

  两次 16 位有符号乘法，然后将 32 位乘积的和或差进行 64 位累加。

- 第4-90 页的*UMAAL*

  无符号长整型乘加累加。

- 第4-91 页的*MIA、MIAPH 和 MIAxy*

  XScale 协处理器 0 指令（内部累加乘法）。

4.5.1 MUL、MLA 和 MLS

乘法、乘加和乘减，有符号或无符号 32 位操作数，结果取低 32 位。

语法

MUL{S}{*cond*} {*Rd*}, *Rn*, *Rm*

MLA{S}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

MLS{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

S 是一个可选的后缀。如果指定了 S，则将会更新运算结果的条件代码标记（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn, *Rm* 是存放乘数的寄存器。

Ra 存放被加数和被减数的寄存器。

用法

MUL 指令可将 *Rn* 和 *Rm* 中的值相乘，并将所得结果的低 32 位 存入 *Rd*。

MLA 指令可将 *Rn* 和 *Rm* 中的值相乘，然后再将乘积与 *Ra* 中的值相加，最后将所得和的低 32 位存入 *Rd*。

MLS 指令可将 *Rn* 和 *Rm* 中的值相乘，然后再从 *Ra* 中的值中减去乘积，最后将所得差的低 32 位存入 *Rd*。

不能将 r15 用作 *Rd*、*Rn*、*Rm* 或 *Ra*。

条件标记

如果指定了 `S`，则 `MUL` 和 `MLA` 指令将:

- 根据结果来更新 N 和 Z 标记。
- 如果是 ARMv4 及更早版本，则破坏 C 和 V 标记
- 如果是 ARMv5 及更高版本，则不影响 C 或 V 标记。

Thumb 指令

`MUL` 指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令:

`MULS` *`Rd`*, *`Rn`*, *`Rd`*　　*`Rd`* 和 *`Rn`* 必须都为 Lo 寄存器。

这是 Thumb 指令中唯一可设置条件代码标记的指令。

体系结构

ARM 指令 `MUL` 和 `MLA` 可用于 ARM 体系结构的所有版本中。

ARM 指令 `MLS` 可用于 ARMv6T2 和 ARMv7。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

`MULS` 16 位 Thumb 指令可用于 ARM 体系结构的所有 T 变体中。

示例

```
    MUL     r10, r2, r5
    MLA     r10, r2, r1, r5
    MULS    r0, r2, r2
    MULLT   r2, r3, r2
    MLS     r4, r5, r6, r7
```

4.5.2 UMULL、UMLAL、SMULL 和 SMLAL

有符号长乘法和无符号长乘法，可选择进行累加，32 位操作数，结果和累加器均为 64 位。

语法

`Op{S}{cond} RdLo, RdHi, Rn, Rm`

其中：

Op 可为 `UMULL`、`UMLAL`、`SMULL` 或 `SMLAL` 之一。

`S` 为一个可选后缀，仅可用于 ARM 状态。如果指定 `S`，则将会更新运算结果的条件代码标记（请参阅第2-17 页的*条件执行*）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

RdLo, *RdHi* 是目标寄存器。对于 `UMLAL` 和 `SMLAL`，它们还用于存放累加值。*RdLo* 和 *RdHi* 必须为不同的寄存器

Rn, *Rm* 是存放操作数的 ARM 寄存器。

不能将 r15 用作 *RdHi*、*RdLo*、*Rn* 或 *Rm*。

用法

`UMULL` 指令会将 *Rn* 和 *Rm* 中的值解释为无符号整数。它会先求这两个整数的乘积，然后将结果的低 32 位存入 *RdLo*，高 32 位存入 *RdHi*。

`UMLAL` 指令会将 *Rn* 和 *Rm* 中的值解释为无符号整数。它会先求这两个整数的乘积，然后将 64 位结果与 *RdHi* 和 *RdLo* 中所包含的 64 位无符号整数相加。

`SMULL` 指令会将 *Rn* 和 *Rm* 中的值解释为有符号整数的二进制补码。它会先求这两个整数的乘积，然后将结果的低 32 位存入 *RdLo*，高 32 位存入 *RdHi*。

`SMLAL` 指令会将 *Rn* 和 *Rm* 中的值解释为有符号整数的二进制补码。它们会先求这两个整数的乘积，然后将 64 位结果与 *RdHi* 和 *RdLo* 中的 64 位有符号整数相加。

条件标记

如果指定了 `S`，则这些指令将：

- 根据结果来更新 N 和 Z 标记。
- 不影响 C 或 V 标记。

体系结构

这些 ARM 指令可用于所有版本的 ARM 体系结构中。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

示例

```
    UMULL       r0, r4, r5, r6
    UMLALS      r4, r5, r3, r8
```

4.5.3 SMUL*xy* 和 SMLA*xy*

有符号乘法和乘加，16 位操作数，结果和累加器均为 32 位。

语法

SMUL<*x*><*y*>{*cond*} {*Rd*}, *Rn*, *Rm*

SMLA<*x*><*y*>{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

其中:

<*x*> 可为 B 或 T。B 意味着使用 *Rm* 的低 16 位（位 [15:0]），T 意味着使用 *Rm* 的高 16 位（[31:16]）。

<*y*> 可为 B 或 T。B 意味着使用 *Rs* 的低 16 位（位 [15:0]），T 意味着使用 *Rs* 的高 16 位（位 [31:16]）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn, *Rm* 存放要相乘的值的寄存器。

Ra 是存放要相加的值的寄存器。

用法

不能将 r15 用作 *Rd*、*Rn*、*Rm* 或 *Ra*。

SMULxy 可将从 *Rn* 和 *Rm* 中选择的 16 位有符号整数相乘，并会将 32 位结果存入 *Rd*。

SMLAxy 可将从 *Rn* 和 *Rm* 中选择的 16 位有符号整数相乘，并会将 32 位结果与 *Ra* 中的 32 位值相加，最后将结果存入 *Rd*。

条件标记

这些指令不影响 N、Z、C 或 V 标记。

如果累加时发生溢出，SMLAxy 将设置 Q 标记。若要读取 Q 标记的状态，请使用 MRS 指令（请参阅第4-131 页的*MRS*）。

注意

SMLAxy 不会清除 Q 标记。要清除 Q 标记，则请使用 MSR 指令（请参阅第4-133 页的*MSR*）。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
    SMULTBEQ    r8, r7, r9
    SMLABBNE    r0, r2, r1, r10
    SMLABT      r0, r0, r3, r5
```

4.5.4 SMULW*y* 和 SMLAW*y*

有符号乘法和有符号乘加，一个 32 位操作数和一个 16 位操作数，结果取高 32 位。

语法

SMULW<*y*>{*cond*} {*Rd*}, *Rn*, *Rm*

SMLAW<*y*>{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

其中:

<*y*> 可为 B 或 T。B 意味着使用 *Rs* 的低 16 位（位 [15:0]），T 意味着使用 *Rs* 的高 16 位（位 [31:16]）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn, *Rm* 存放要相乘的值的寄存器。

Ra 是存放要相加的值的寄存器。

用法

不能将 r15 用作 *Rd*、*Rn*、*Rm* 或 *Ra*。

SMULWy 可将选自 *Rm* 的 16 位有符号整数与 *Rn* 中的有符号整数相乘，并将 48 位结果的高 32 位存入 *Rd*。

SMLAWy 可将选自 *Rm* 的 16 位有符号整数与 *Rn* 中的值相乘，然后将 32 位结果与 *Ra* 中的 32 位值相加，最后将结果存入 *Rd*。

条件标记

这些指令不更改标记。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

4.5.5 SMLAL*xy*

有符号乘加，16 位操作数，64 位累加器。

语法

SMLAL<*x*><*y*>{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

其中：

<*x*> 可为 B 或 T。B 意味着使用 *Rm* 的低 16 位（位 [15:0]），T 意味着使用 *Rm* 的高 16 位（[31:16]）。

<*y*> 可为 B 或 T。B 意味着使用 *Rs* 的低 16 位（位 [15:0]），T 意味着使用 *Rs* 的高 16 位（位 [31:16]）。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

RdHi, *RdLo* 是目标寄存器。它们也存放累加值。*RdHi* 和 *RdLo* 必须为不同的寄存器。

Rn, *Rm* 存放要相乘的值的寄存器。

不能将 r15 用作 *RdHi*、*RdLo*、*Rn* 或 *Rm*。

用法

SMLALxy 可将选自 *Rm* 的 16 位有符号整数与选自 *Rn* 的 16 位有符号整数相乘，然后将 32 位乘积结果与 *RdHi* 和 *RdLo* 中的 64 位值相加。

条件标记

此指令不更改标记。

——注意——

SMLALxy 不会产生异常。如果此指令发生溢出，则将不返回结果，且不发出警告。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本和 ARMv5 的 E 变体中。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

此指令无 16 位 Thumb 版本。

示例

```
    SMLALTB     r2, r3, r7, r1
    SMLALBTVS   r0, r1, r9, r2
```

4.5.6 SMUAD{X} 和 SMUSD{X}

两次 16 位有符号乘法，然后将两个乘积相加或相减，可选择交换操作数的高半字和低半字。

语法

`op{X}{cond} {Rd}, Rn, Rm`

其中：

op 为下列项之一：

`SMUAD` 两次乘法，然后将乘积相加。

`SMUSD` 两次乘法，然后将乘积相减。

`X` 是一个可选的参数。如果有 `X`，则在相乘之前，会先交换第二个操作数的高半字和低半字。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn, *Rm* 是存放操作数的寄存器。

不能将 r15 用作 *Rd*、*Rn* 或 *Rm*。

用法

`SMUAD` 可将 *Rn* 的低半字与 *Rm* 的低半字相乘，*Rn* 的高半字与 *Rm* 的高半字相乘。然后，将两个乘积相加，并将结果存入 *Rd*。

`SMUSD` 可将 *Rn* 的低半字与 *Rm* 的低半字相乘，*Rn* 的高半字与 *Rm* 的高半字相乘。然后，从第一个乘积中减去第二个乘积，并将差值存入到 *Rd*。

条件标记

如果进行加法运算时溢出，`SMUAD` 指令将设置 Q 标记。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
    SMUAD       r2, r3, r2
    SMUSDXNE    r0, r1, r2
```

4.5.7 SMMUL、SMMLA 和 SMMLS

有符号高字乘法、有符号高字乘加和有符号高字乘减。这些指令的操作数为 32 位，结果仅取高 32 位。

语法

```
SMMUL{R}{cond} {Rd}, Rn, Rm
SMMLS{R}{cond} Rd, Rn, Rm, Ra
SMMLS{R}{cond} Rd, Rn, Rm, Ra
```

其中:

R　　是一个可选的参数。如果存在 R，则对结果进行舍入，否则将其截断。

cond　　是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd　　是目标寄存器。

Rn, *Rm*　　是存放操作数的寄存器。

Ra　　存放被加数和被减数的寄存器。

不能将 r15 用作 *Rd*、*Rn*、*Rm* 或 *Ra*。

操作

SMMUL 可将 *Rn* 和 *Rm* 中的值相乘，然后将 64 位结果的高 32 位存入 *Rd*。

SMMLA 可将 *Rn* 和 *Rm* 中的值相乘，然后将 *Ra* 中的值与乘积的高 32 位相加，最后将结果存入 *Rd*。

SMMLS 可将 *Rn* 和 *Rm* 中的值相乘，接着将 *Ra* 中的值左移 32，然后从移位后的值中减去乘积，最后将差的高 32 位存入 *Rd*。

如果指定了可选参数 R，则在截取结果的高 32 前，会先加上 0x80000000。这对结果的舍入有影响。

条件标记

这些指令不更改标记。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
SMMULGE     r6, r4, r3
SMMULR      r2, r2, r2
```

4.5.8 SMLAD 和 SMLSD

两次 16 位有符号乘法，然后将乘积相加或相减，32 位累加。

语法

op{X}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

其中：

op 为下列项之一：

SMLAD 两次乘法，累加乘积的和。

SMLSD 两次乘法，累加乘积的差。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

X 是一个可选的参数。如果有 X，则在相乘之前，会先交换第二个操作数的高半字和低半字。

Rd 是目标寄存器。

Rn, *Rm* 是存放操作数的寄存器。

Ra 是存放累加操作数的寄存器。

不能将 r15 用作 *Rd*、*Rn*、*Rm* 或 *Ra*。

操作

SMLAD 可将 *Rn* 的低半字与 *Rm* 的低半字相乘，将 *Rn* 的高半字与 *Rm* 的高半字相乘。然后，将两个乘积与 *Ra* 中的值相加，并将和存入 *Rd*。

SMLSD 可将 *Rn* 的低半字与 *Rm* 的低半字相乘，将 *Rn* 的高半字与 *Rm* 的高半字相乘。然后，从第一个乘积中减去第二个乘积，接着将所得的差与 *Ra* 中的值相加，最后将结果存入 *Rd*。

条件标记

这些指令不更改标记。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
    SMLSD       r1, r2, r0, r7
    SMLSDX      r11, r10, r2, r3
    SMLADLT     r1, r2, r4, r1
```


4.5.9 SMLALD 和 SMLSLD

两次 16 位有符号乘法，然后将乘积相加或相减，64 位累加。

语法

op{X}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

其中：

op 是下列项之一：

- SMLALD 两次乘法，累加乘积的和。
- SMLSLD 两次乘法，累加乘积的差。

X 是一个可选的参数。如果有 X，则在相乘之前，会先交换第二个操作数的高半字和低半字。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

RdLo, *RdHi* 是存放 64 位结果的目标寄存器。它们还要存放 64 位累加操作数。*RdHi* 和 *RdLo* 必须为不同的寄存器。

Rn, *Rm* 是存放操作数的寄存器。

不能将 r15 用作 *RdLo*、*RdHi*、*Rn* 或 *Rm*。

操作

SMLALD 可将 *Rn* 的低半字与 *Rm* 的低半字相乘，将 *Rn* 的高半字与 *Rm* 的高半字相乘。然后，将两个乘积与 *RdLo* 和 *RdHi* 中的值相加，并将所得的和存入 *RdLo*、*RdHi* 中。

SMLSLD 可将 *Rn* 的低半字与 *Rm* 的低半字相乘，将 *Rn* 的高半字与 *Rm* 的高半字相乘。然后，从第一个乘积中减去第二个乘积，将所得的差与 *RdLo*、*RdHi* 中的值相加，并把结果存入 *RdLo*、*RdHi*。

条件标记

这些指令不更改标记。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
    SMLALD      r10, r11, r5, r1
    SMLSLD      r3, r0, r5, r1
```

4.5.10 UMAAL

无符号长整型乘加累加。

语法

UMAAL{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

RdLo, *RdHi* 是存放 64 位结果的目标寄存器。它们还要存放两个 32 位累加操作数。*RdLo* 和 *RdHi* 必须为不同的寄存器。

Rn, *Rm* 为存为乘法操作数的寄存器。

不能将 r15 用作 *RdLo*、*RdHi*、*Rn* 或 *Rm*。

操作

UMAAL 指令可将 *Rn* 和 *Rm* 中的 32 位值相乘，然后再将乘积与 *RdHi* 和 *RdLo* 中的两个 32 位值相加，最后将 64 位结果存入 *RdLo*，*RdHi*。

条件标记

此指令不更改标记。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本和 ARMv5 的 E 变体中。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

此指令无 16 位 Thumb 版本。

示例

```
    UMAAL       r8, r9, r2, r3
    UMAALGE     r2, r0, r5, r3
```


4.5.11 MIA、MIAPH 和 MIAxy

XScale 协处理器 0 指令。

带内部累加的乘法（32 位乘 32 位，40 位累加）。

带内部累加的乘法，组合半字（16 位乘 16 位两次，40 位累加）。

带内部累加的乘法（16 位乘 16 位，40 位累加）。

语法

```
MIA{cond} Acc, Rn, Rm
```

```
MIAPH{cond} Acc, Rn, Rm
```

```
MIA<x><y>{cond} Acc, Rn, Rm
```

其中：

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Acc 是内部累加器。此累加器的标准名称为 accx，其中 *x* 为一个整数，范围为 0 到 *n*。具体的 *n* 值取决于处理器。在目前的处理器中它是 0。

Rn, *Rm* 是存放要相乘的值的 ARM 寄存器。

不能将 r15 用作 *Rn* 或 *Rm*。

<x><y> 可为 BB、BT、TB 或 TT 之一。

用法

MIA 指令可将 *Rn* 和 *Rm* 中的两个有符号整数值相乘，并会将乘积加到 *Acc* 中的 40 位值上。

MIAPH 指令可将 *Rn* 和 *Rm* 的低半字有符号整数相乘，将 *Rn* 和 *Rm* 的高半字整数相乘，然后再将两个 32 位乘积与 *Acc* 中的 40 值相加。

MIAxy 指令可将选自 *Rs* 的 16 位有符号整数与选自 *Rm* 的 16 位有符号整数相乘，然后再将 32 位乘积与 *Acc* 中的 40 位值相加。*<x>* == B 表示使用 *Rn* 的低 16 位（位 [15:0]），*<x>* == T 表示使用 *Rn* 的高 16 位（位 [31:16]）。*<y>* == B 表示使用 *Rm* 的低 16 位（位 [15:0]），*<y>* == T 表示使用 *Rm* 的高 16 位（位 [31:16]）。

条件标记

这些指令不更改标记。

——注意——

这些指令不会产生异常。如果这些指令发生溢出，则将不返回结果，且不发出警告。

体系结构

这些 ARM 指令只可用于 XScale 处理器。

这些指令没有 Thumb 版本。

示例

```
    MIA     acc0,r5,r0
    MIALE   acc0,r1,r9
    MIAPH   acc0,r0,r7
    MIAPHNE acc0,r11,r10
    MIABB   acc0,r8,r9
    MIABT   acc0,r8,r8
    MIATB   acc0,r5,r3
    MIATT   acc0,r0,r6
    MIABTGT acc0,r2,r5
```

4.6 饱和指令

本节包括以下小节:
- *饱和算法*
- 第4-94 页的*QADD、QSUB、QDADD 和 QDSUB*
- 第4-96 页的*SSAT 和 USAT*。

有些并行指令同时也是饱和指令，请参阅第4-98 页的*并行指令*。

4.6.1 饱和算法

这些指令均为*饱和的*（SAT）。这意味着对于某些依赖于指令 2^n 相关值:

- 对于有符号饱和运算，如果结果小于 -2^n，则返回的结果将为 -2^n
- 对于无符号饱和运算，如果整个结果将是负值，那么返回的结果是 0
- 如果结果大于 $2^n - 1$，则返回的结果将为 $2^n - 1$。

只要出现这些情况，就称为*饱和*。当出现饱和时，有些指令会设置 Q 标记。

——注意——

当未出现饱和时，饱和指令不清除 Q 标记。要清除 Q 标记，则请使用 `MSR` 指令（请参阅第4-133 页的*MSR*）。

您也可以利用其他两个指令来设置 Q 标记（请参阅第4-77 页的*SMULxy 和 SMLAxy* 和 第4-79 页的*SMULWy 和 SMLAWy*），这些指令不是饱和指令。

4.6.2 QADD、QSUB、QDADD 和 QDSUB

有符号加法、减法，加倍加法，加倍减法，饱和到有符号范围 $-2^{31} \le x \le 2^{31}-1$ 内。

另请参阅第4-99 页的*并行加法和减法*。

语法

```
op{cond} {Rd}, Rm, Rn
```

其中:

op 可为 QADD、QSUB、QDADD 或 QDSUB 之一。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rm、*Rn* 是存放操作数的寄存器。

不能将 r15 用作 *Rd*、*Rm* 或 *Rn*。

用法

QADD 指令可将 *Rm* 和 *Rn* 中的值相加。

QSUB 指令可从 *Rm* 中的值中减去 *Rn* 中的值。

QDADD 指令可计算 SAT(*Rm* + SAT(*Rn* * 2))。进行加倍和加法运算均有可能出现饱和。如果加倍运算发生饱和，而加法运算没有出现饱和，则将设置 Q 标记，但最终结果是不饱和的。

QDSUB 指令可计算 SAT(*Rm* - SAT(*Rn* * 2))。进行加倍和加法运算均有可能出现饱和。如果加倍运算发生饱和，而加法运算没有出现饱和，则将设置 Q 标记，但最终结果是不饱和的。

——注意——

这些指令会将所有值视为有符号整数的二进制补码。

有关仅可在 ARMv6 及更高版本中可用的类似指令的信息，请参阅第4-99 页的*并行加法和减法*。

条件标记

如果发生饱和，这些指令设置 Q 标记。若要读取 Q 标记的状态，请使用 `MRS` 指令（请参阅第4-131 页的*MRS*）。

体系结构

这些指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
QADD    r0, r1, r9
QDSUBLT r9, r0, r1
```

4.6.3 SSAT 和 USAT

有符号饱和到任何位位置和无符号饱和到任何位位置，可选择在饱和前进行移位。

SSAT 将有符号值饱和到有符号范围内。

USAT 可将有符号值饱和到无符号范围内。

另请参阅第4-104 页的*SSAT16 和 USAT16*。

语法

op{*cond*} *Rd*, #*sat*, *Rm*{, *shift*}

其中:

op 可为 SSAT 或 USAT。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 为目标寄存器。*Rd* 不能为 r15。

sat 指定要饱和到的位位置，SSAT 的范围在 1 到 32 之间，USAT 的范围在 0 到 31 之间。

Rm 为存放操作数的寄存器。*Rm* 不能为 r15。

shift 是一个可选的移位。必须为下列项之一：

- ASR #*n* 其中，*n* 的范围为 1-32 (ARM) 或 1-31 (Thumb-2)
- LSL #*n* 其中，*n* 的范围为 0-31。

操作

SSAT 指令会先进行指定的移位，然后将结果饱和到有符号范围 $-2^{sat-1} \leq x \leq 2^{sat-1} -1$。

USAT 指令会先进行指定的移位，然后饱和到无符号范围 $0 \leq x \leq 2^{sat} - 1$。

条件标记

如果发生饱和，这些指令设置 Q 标记。若要读取 Q 标记的状态，请使用 MRS 指令（请参阅第4-131 页的*MRS*）。

体系结构

这些 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

示例

```
SSAT    r7, #16, r7, LSL #4
USATNE  r0, #7, r5
```

4.7 并行指令

本节包括以下小节:

- 第4-99 页的*并行加法和减法*

  多种字节和半字加法和减法。

- 第4-102 页的*USAD8 和 USADA8*

  无符号值的差的绝对值求和，无符号值的差的绝对值求和累加。

- 第4-104 页的*SSAT16 和 USAT16*

  并行半字饱和指令。

除此之外，还有其他一些并行分离指令，请参阅第4-109 页的*SXT*、*SXTA*、*UXT* *和 UXTA*。

4.7.1 并行加法和减法

多种字节和半字加法和减法。

语法

<prefix>op{*cond*} {*Rd*}, *Rn*, *Rm*

其中:

<prefix> 是下列项之一:

S	对 2^8 或 2^{16} 有符号求模。设置 APSR GE 标记。
Q	有符号饱和算法。
SH	有符号算法，将结果减半。
U	对 2^8 或 2^{16} 无符号求模。设置 APSR GE 标记。
UQ	无符号饱和算法。
UH	无符号算法，将结果减半。

op 是下列项之一:

ADD8	以字节为单位的加法
ADD16	以半字为单位的加法。
SUB8	以字节为单位的减法。
SUB16	以半字为单位的减法。
ASX	先交换 *Rm* 的半字，然后将高半字相加，接着将低半字相减。
SAX	交换 *Rm* 的半字，然后将高半字相减，将低半字相加。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。不能将 r15 用作 *Rd*。

Rm、*Rn* 是存放操作数的 ARM 寄存器。不能将 r15 用作 *Rm* 或 *Rn*。

操作

这些指令可对操作数的字节或半字进行单独运算。它们可执行两次或四次加法或减法，或一次加法和一次减法。

您可以选择各种算术运算:

- 对 2^8 或 2^{16} 进行有符号或无符号算术求模。这将会设置 APSR GE 标记（请参阅*条件标记*）。
- 有符号饱和到有符号范围 $-2^{15} \le x \le 2^{15}-1$ or $-2^7 \le x \le 2^7-1$ 内。即使这些运算饱和，Q 标记也不受影响。
- 无符号饱和到无符号范围 $0 \le x \le 2^{16}-1$ or $0 \le x \le 2^8-1$ 内。即使这些运算饱和，Q 标记也不受影响。
- 有符号或无符号算术运算，将结果减半。这种运算不会造成溢出。

条件标记

这些指令不影响 N、Z、C、V 或 Q 标记。

这些指令的前缀变体 `Q`、`SH`、`UQ` 和 `UH` 不更改标记。

这些指令的前缀变体 `S` 和 `U` 会设置 APSR 中的 GE 标记，如下所述:

- 对于以字节为单位的运算，GE 标记的用法与 32 位 `SUB` and `ADD` 指令的 C（进位）标记相同:

  `GE[0]` 用于结果的 [7:0] 位

  `GE[1]` 用于结果的 [15:8] 位

  `GE[2]` 用于结果的 [23:16] 位

  `GE[3]` 用于结果的 [31:24] 位

- 对于以半字为单位的运算，GE 标记的用法与以字为单位的 `SUB` 和 `ADD` 指令的 C（进位）标记相同

  `GE[1:0]` 用于结果的 [15:0] 位

  `GE[3:2]` 用于结果的 [31:16] 位

您可利用这些标记来控制后续的 `SEL` 指令，请参阅第4-62 页的*SEL*。

——注意——

对于半字方式的运算，将同时设置或清除 GE[1:0]，并同时设置或清除 GE[3:2]。

————————

体系结构

这些 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
    SHADD8      r4, r3, r9
    USAXNE      r0, r0, r2
```

不正确的示例

```
    QHADD       r2, r9, r3    ; No such instruction, should be QHADD8 or QHADD16
    SAX         r10, r8, r5   ; Must have a prefix.
```

4.7.2 USAD8 和 USADA8

无符号值的差的绝对值求和，无符号值的差的绝对值求和累加。

语法

USAD8{*cond*} {*Rd*}, *Rn*, *Rm*

USADA8{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn 是存放第一个操作数的寄存器。

Rm 是存放第二个操作数的寄存器。

Ra 是存放累加操作数的寄存器。

不能将 r15 用作 *Rd*、*Rn*、*Rn* 或 *Ra*。

操作

USAD8 指令可计算 *Rn* 和 *Rm* 的四个对应字节所含值之间的差。然后，它会将这四个差值的绝对值相加，并将结果存入 *Rd*。

USADA8 指令可将四个差值的绝对值加到 *Ra* 的值上，并将结果存入 *Rd*。

条件标记

这些指令不更改任何标记。

体系结构

这些 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
USAD8       r2, r4, r6
USADA8      r0, r3, r5, r2
USADA8VS    r0, r4, r0, r1
```

不正确的示例

```
USADA8      r2, r4, r6      ; USADA8 requires four registers
USADA16     r0, r4, r0, r1  ; no such instruction
```

4.7.3 SSAT16 和 USAT16

并行半字饱和指令。

SSAT16 可将有符号值饱和到有符号范围内。

USAT16 可将有符号值饱和到无符号范围内。

语法

op{*cond*} *Rd*, #*sat*, *Rn*

其中:

op 是下列项之一:

- SSAT16 有符号饱和。
- USAT16 无符号饱和。

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。

Rd 是目标寄存器。

sat 指定要饱和到的位位置，SSAT16 的范围在 1 到 16 之间，USAT16 的范围在 0 到 15 之间。

Rn 是存放操作数的寄存器。

不要将 r15 用作 *Rd* 或 *Rn*。

操作

有符号和无符号半字饱和任何位位置。

SSAT16 指令可将每个有符号半字饱和到有符号范围 $-2^{sat-1} \leq x \leq 2^{sat-1} -1$ 内。

USAT16 指令可将每个有符号半字饱和到无符号范围 $0 \leq x \leq 2^{sat} -1$ 内。

条件标记

只要有半字发生饱和，则这些指令就会设置 Q 标记。若要读取 Q 标记的状态，请使用 MRS 指令(请参阅第4-131 页的*MRS*)。

体系结构

这些 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
SSAT16  r7, #12, r7
USAT16  r0, #7, r5
```

不正确的示例

```
SSAT16  r1, #16, r2, LSL #4 ; shifts not permitted with halfword saturations
```

4.8 组合和分离指令

本节包括以下小节:

- 第4-107 页的*BFC 和 BFI*

  位域清零和位域插入
- 第4-108 页的*SBFX 和 UBFX*

  有符号或无符号位域提取。
- 第4-109 页的*SXT*、*SXTA*、*UXT 和 UXTA*

  符号扩展或零扩展指令，可选择进行加法运算。
- 第4-112 页的*PKHBT 和 PKHTB*

  半字组合指令。

4.8.1 BFC 和 BFI

位域清零和位域插入 清除寄存器中相邻位，或将一个寄存器中的相邻位插入另一个寄存器中的相邻位中。

语法

BFC{*cond*} *Rd*, #*lsb*, #*width*

BFI{*cond*} *Rd*, *Rn*, #*lsb*, #*width*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 为目标寄存器。*Rd* 不能为 r15。

Rn 为源寄存器。*Rn* 不可为 r15。

lsb 是要清除或复制的最低有效位。

width 是要清除或复制的位的数量。*width* 一定不能为 0，并且 (*width*+*lsb*) 必须小于 32。

BFC

从 *lsb* 开始，清除 *Rd* 中的 *width* 个位。*Rd* 中的其他位保持不变。

BFI

用从 [0] 位开始的 *Rn* 中的 *width* 位替换 *Rd* 中的从 *lsb* 开始的 *width* 位。*Rd* 中的其他位保持不变。

条件标记

这些指令不更改标记。

体系结构

这些 ARM 指令可用于 ARMv6T2 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

4.8.2 SBFX 和 UBFX

有符号和无符号位域提取。将一个寄存器的相邻的位复制到另一个寄存器的最低有效位，并用符号或零扩展到 32 位。

语法

op{*cond*} *Rd*, *Rn*, #*lsb*, #*width*

其中:

op 是 SBFX 或 UBFX。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。

Rn 是源寄存器。

lsb 是位域中的最低有效位的位编码，范围从 0 到 31。

width 是位域宽度，范围从 1 到 (32-*lsb*)。

不要将 r15 用作 *Rd* 或 *Rn*。

条件标记

这些指令不更改任何标记。

体系结构

这些 ARM 指令可用于 ARMv6T2 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

4.8.3 SXT、SXTA、UXT 和 UXTA

符号扩展、带加法的符号扩展、零扩展和带加法的零扩展。

语法

```
SXT<extend>{cond} {Rd}, Rm{, rotation}
SXTA<extend>{cond} {Rd}, Rn, Rm{, rotation}
UXT<extend>{cond} {Rd}, Rm{, rotation}
UXTA<extend>{cond} {Rd}, Rn, Rm{, rotation}
```

其中:

<extend> 是下列项之一:

- `B16` 将两个 8 位值扩展为两个 16 位值。
- `B` 将一个 8 位值扩展为一个 32 位值。
- `H` 将一个 16 位值扩展为一个 32 位值。

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。

Rd 是目标寄存器。

Rn 是存放要相加的值的寄存器(仅 `SXTA` 和 `UXTA`)。

Rm 是存放要扩展的值的寄存器。

rotation 是下列项之一:

- `ROR #8` 将 *Rm* 中的值向右循环移 8 位。
- `ROR #16` 将 *Rm* 中的值向右循环移 16 位。
- `ROR #24` 将 *Rm* 中的值向右循环移 24 位。

如果省略 *rotation*,则不执行循环。

不能将 r15 用作 *Rd*、*Rn* 或 *Rm*。

操作

这些指令执行以下操作:

1. 将 *Rm* 中的值向右循环移 0、8、16 或 24 位(仅适用于 ARM 和 Thumb-2)。
2. 对所得到的值执行下列操作:
   - 提取 [7:0] 位,并符号或零扩展到 32 位。如果指令为扩展并相加指令,则加上 *Rn* 中的值。
   - 提取 [15:0] 位,并符号或零扩展到 32 位。如果指令为扩展并相加指令,则加上 *Rn* 中的值。
   - 提取 [23:16] 位以及 [7:0] 位,并符号或零扩展到 16 位。如果指令是扩展并相加,则分别将它们加到 *Rn* 的 [31:16] 位和 [15:0] 位上,以形成结果的 [31:16] 位和 [15:0] 位。

条件标记

这些指令不更改标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中,在 Thumb-2 代码中使用时为 16 位指令:

`SXTB` *Rd*, *Rm*	*Rd* 和 *Rm* 必须都为 Lo 寄存器。
`SXTH` *Rd*, *Rm*	*Rd* 和 *Rm* 必须都为 Lo 寄存器。
`UXTB` *Rd*, *Rm*	*Rd* 和 *Rm* 必须都为 Lo 寄存器。
`UXTH` *Rd*, *Rm*	*Rd* 和 *Rm* 必须都为 Lo 寄存器。

体系结构

这些 ARM 指令可用于 ARMv6 及更高版本。

`SXT` 和 `UXT` Thumb 指令可用于 ARMv6T2 和 ARMv7。

`SXTA` 和 `UXTA` Thumb 指令可用于 ARMv6T2 和 ARMv7,但 ARMv7-M 架构要除外。

这些 16 位 Thumb 指令可用于 ARMv6 及更高版本。

示例

```
SXTH        r3, r9, r4
UXTAB16EQ   r0, r0, r4, ROR #16
```

不正确的示例

```
SXTH    r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
```

4.8.4 PKHBT 和 PKHTB

半字组合指令。

组合两个寄存器中的半字。可在提取半字之前，对其中一个操作数进行移位。

语法

```
PKHBT{cond} {Rd}, Rn, Rm{, LSL #leftshift}
PKHTB{cond} {Rd}, Rn, Rm{, ASR #rightshift}
```

其中：

PKHBT　将 *Rn* 的位 [15:0] 与移位后的 *Rm* 的位 [31:16] 进行组合。

PKHTB　将 *Rn* 的位 [31:16] 与移位后的 *Rm* 的位 [15:0] 进行组合。

cond　是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd　是目标寄存器。

Rn　是存放第一个操作数的寄存器。

Rm　是存放第一个操作数的寄存器。

leftshift　在 0 到 31 范围内。

rightshift　在 1 到 32 范围内。

不能将 r15 用作 *Rd*、*Rn* 或 *Rm*。

条件标记

这些指令不更改标记。

体系结构

这些 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7，但 ARMv7-M 架构要除外。

这些指令均无 16 位 Thumb 版本。

示例

```
PKHBT   r0, r3, r5          ; combine the bottom halfword of r3 with
                              the top halfword of r5
PKHBT   r0, r3, r5, LSL #16 ; combine the bottom halfword of r3 with
                              the bottom halfword of r5
PKHTB   r0, r3, r5, ASR #16 ; combine the top halfword of r3 with
                              the top halfword of r5
```

您还可通过使用不同的移位值来调整第二个操作数。

不正确的示例

```
PKHBTEQ r4, r5, r1, ASR #8  ; ASR not permitted with PKHBT
```


4.9 跳转指令

本节包括以下小节:

- 第4-115 页的*B、BL、BX、BLX 和 BXJ*

  跳转、带链接的跳转、跳转并交换指令集、带链接的跳转并交换指令集、跳转并将指令集转换到 Jazelle。

- 第4-118 页的*CBZ 和 CBNZ*

  与零比较并跳转。

- 第4-119 页的*TBB 和 TBH*

  表跳转字节或半字。

4.9.1 B、BL、BX、BLX 和 BXJ

跳转、带链接的跳转、跳转并交换指令集、带链接的跳转并交换指令集、跳转并转换到 Jazelle 状态。

语法

`op{cond}{.W} label`

`op{cond} Rm`

其中:

op 是下列项之一:

B	跳转。
BL	带链接的跳转。
BX	跳转并交换指令集。
BLX	带链接跳转并交换指令集。
BXJ	跳转并转换为 Jazelle 执行。

cond 是一种可选的条件代码(请参阅第2-17 页的*条件执行*)。*cond* 不适用于所有形式的这种指令,请参阅第4-116 页的*指令可用性和跳转范围*。

.W 是一个可选的指令宽度说明符,用于强制要求在 Thumb-2 中使用 32 位 B 指令。有关详细信息,请参阅第4-117 页的*Thumb-2 中的 B*。

label 是一个程序相对的表达式。有关详细信息,请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

Rm 是一个寄存器,包含要跳转到的目标地址。

操作

所有这些指令均会引发跳转,或跳转到 *label*,或跳转到包含在 *Rm* 中的地址处。此外:

- BL 和 BLX 指令可将下一个指令的地址复制到 r14(lr,链接寄存器)中。
- BX 和 BLX 指令可将处理器的状态从 ARM 更改为 Thumb,或从 Thumb 更改为 ARM。

  无论何种情况,BLX *label* 始终会更改处理器的状态。

BX *Rm* 和 BLX *Rm* 可从 *Rm* 的位 [0] 推算出目标状态

— 如果 *Rm* 的位 [0] 为 0，则处理器的状态将改为（或保持在）ARM 状态

— 如果 *Rm* 的位 [0] 为 1，则处理器的状态将更改（或保持）为 Thumb 状态。

- BXJ 指令会将处理器的状态更改为 Jazelle。

指令可用性和跳转范围

表4-7 给出了可在 ARM 和 Thumb 状态下使用的指令。此表中未列出的指令不可在这两种状态下使用。括号中的注释给出了第一个可在其中使用指令的体系结构版本。

表4-7 跳转指令的可用性和范围

指令	ARM		16 位 Thumb		32 位 Thumb
B label	±32MB	（所有）	±2KB	（所有 T）	±16MB[a]
B{cond} label	±32MB	（所有）	-252 到 +258	（所有 T）	±1MB[a]
B Rm	使用 BX Rm		使用 BX Rm		使用 16 位 BX Rm
B{cond} Rm	使用 BX{cond} Rm		-		-
BL label	±32MB	（所有）	±4MB[b]	（所有 T）	±16MB
BL{cond} label	±32MB	（所有）	-		-
BL Rm	使用 BLX Rm		使用 BLX Rm		使用 16 位 BLX Rm
BL{cond} Rm	使用 BLX{cond} Rm		-		-
BX Rm	可用	（4T，5）	可用	（所有 T）	使用 16 位
BX{cond} Rm	可用	（4T，5）	-		-
BLX label	±32MB	(5)	±4MB[c]	（5T）	±16MB [d]
BLX Rm	可用	(5)	可用	（5T）	使用 16 位
BLX{cond} Rm	可用	(5)	-		-
BXJ Rm	可用	（5J，6）	-		可用[d]
BXJ{cond} Rm	可用	（5J，6）	-		-

a. 使用 .W 指示汇编程序使用此 32 位指令。

b. 这是一个指令对。
c. 这是一个指令对。
d. 在 ARMv7-M 中不可用。

扩展跳转范围

机器级指令 B 和 BL 对当前指令有严格的地址范围限制。但是，即使 *label* 超出范围，仍可以使用这些指令。通常您并不知道链接器会将 *label* 放在何处。但必要时，链接器会添加代码，以允许更长的跳转（请参阅 *《RealView 编译工具链接器和实用程序指南》*中的第 3 章 *使用基本链接器功能*）。所添加的代码称为*胶合代码*。

Thumb-2 中的 B

您可以使用 .W 宽度说明符强制 B 在 Thumb-2 代码中生成 32 位指令。

B.W 始终生成 32 位指令，即使使用 16 位指令就可访问目标。

对于向前引用，不带 .W 的 B 始终在 Thumb 代码中生成 16 位指令，即使这会导致无法访问要用 32 位 Thumb 指令访问的目标。

Thumb-2EE 中的 BX、BLX 和 BXJ

这些指令可在 Thumb-2EE 代码中用作跳转，但不能用于更改状态。您不能在 Thumb-2EE 中使用这些指令的 *op*{*cond*} *label* 格式。在寄存器格式中，*Rm* 的位 [0] 必须是 1，代码会在 ThumbEE 状态下，在目标地址继续执行。

条件标记

这些指令不更改标记。

体系结构

有关在每种体系结构中，这些指令的可用性的详细信息，请参阅第4-116 页的*指令可用性和跳转范围*。

示例

```
B       loopA
BLE     ng+8
BL      subC
BLLT    rtX
BEQ     {pc}+4  ; #0x8004
```

4.9.2 CBZ 和 CBNZ

比较，为零则跳转；比较，为非零则跳转。

语法

CBZ *Rn*, *label*

CBNZ *Rn*, *label*

其中:

Rn 是存放操作数的寄存器。

label 是跳转目标。

用法

您可以使用 CBZ 或 CBNZ 指令来保存与零比较指令和跳转代码序列。

除了不更改条件代码标记外，CBZ Rn, label 与下列指令序列功能相同:

```
CMP     Rn, #0
BEQ     label
```

除了不更改条件代码标记外，CBNZ Rn, label 与下列指令序列功能相同:

```
CMP     Rn, #0
BNE     label
```

限制

跳转目标必须在指令之后的 4 到 130 个字节之内。

这些指令一定不能在 IT 块内使用。

条件标记

这些指令不更改标记。

体系结构

这些 16 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令没有 ARM 或 32 位 Thumb 版本。

4.9.3 TBB 和 TBH

表跳转字节或表跳转半字。

语法

```
TBB [Rn, Rm]
```

```
TBH [Rn, Rm, LSL #1]
```

其中:

Rn 是基址寄存器。此寄存器用于存放跳转长度表的地址。

如果将 *Rn* 指定为 r15，则所用的值将为指令地址加 4。

Rm 是索引寄存器。用于存放到跳转长度表的索引。

Rm 不能为 r15。

操作

这些指令可利用单字节偏移表 (TBB) 或半字偏移表 (TBH) 来产生PC 相对的向前跳转。*Rn* 可提供指向表的指针，而 *Rm* 可提供指向表的索引。跳转长度是从表返回的字节值 (TBB) 或半字值 (TBH) 的两倍。

注释

在 Thumb-2EE 中，如果基址寄存器中的值为零，则代码将跳转到位于 HandlerBase - 4 的 NullCheck 处理程序处执行。

体系结构

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令没有 ARM 或 16 位 Thumb 版本。

4.10 协处理器指令

本节不介绍 VFP（请参阅第 5 章 *NEON 和 VFP 编程*）和无线 MMX™ 技术指令（请参阅第 6 章 *无线 MMX 技术指令*）。特定于 XScale 的指令将在本章后面介绍（请参阅第4-128 页的*其他指令*）。

它包含以下几节:

- 第4-121 页的*CDP 和 CDP2*

  协处理器数据操作。

- 第4-122 页的*MCR*、*MCR2*、*MCRR 和 MCRR2*

  从 ARM 寄存器或寄存器移动到协处理器，可带有协处理器操作。

- 第4-124 页的*MRC*、*MRC2*、*MRRC 和 MRRC2*

  从协处理器移动到 ARM 寄存器或寄存器，可带有协处理器操作。

- 第4-126 页的*LDC*、*LDC2*、*STC 和 STC2*

  在内存和协处理器之间传送数据。

——注意——

当所指定的协处理器不存在或被禁用时，协处理器指令可能会引发未定义的指令异常。

4.10.1 CDP 和 CDP2

协处理器数据操作。

语法

op{*cond*} *coproc*, #*opcode1*, *CRd*, *CRn*, *CRm*{, #*opcode2*}

其中:

op	是 CDP 或 CDP2。
cond	是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。在 ARM 代码中，不允许 CDP2 使用 *cond*。
coproc	是要运行指令的协处理器的名称。标准名称为 p*n*，其中 *n* 为 0 到 15 范围内的整数。
opcode1	是一个协处理器特定的操作码。
CRd, *CRn*, *CRm*	是协处理器寄存器。
opcode2	是一个依协处理器而定的可选操作码。

用法

这些指令的具体用法取决于协处理器。有关详细信息，请参阅协处理器文档。

体系结构

CDP ARM 指令在所有版本的 ARM 体系结构中都有效。

CDP2 ARM 指令可用于 ARMv5 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

4.10.2 MCR、MCR2、MCRR 和 MCRR2

从 ARM 寄存器移到协处理器。对于不同的协处理器，有多种附加操作可供您选用。

语法

op{*cond*} *coproc*, #*opcode1*, *Rt*, *CRn*, *CRm*{, #*opcode2*}

op{*cond*} *coproc*, #*opcode1*, *Rt*, *Rt2*, *CRm*

其中:

op 是 MCR、MCR2、MCRR 或 MCRR2 之一。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。在 ARM 代码中，MCR2 或 MCRR2 不允许使用 *cond*。

coproc 是要运行指令的协处理器的名称。标准名称为 p*n*，其中 *n* 为 0 到 15 范围内的整数。

opcode1 是一个协处理器特定的操作码。

Rt, *Rt2* 是 ARM 源寄存器。*Rt2* 仅适用于 MCRR 和 MCRR2。*Rt* 以及 MCRR 或 MCRR2 中的 *Rt2* 都不能使用 r15。

CRn, *CRm* 是协处理器寄存器。*CRn* 仅适用于 MCR 和 MCR2。

opcode2 是一个依协处理器而定的可选操作码，它适用于 MCR 和 MCR2。

用法

这些指令的具体用法取决于协处理器。有关详细信息，请参阅协处理器文档。

体系结构

`MCR` ARM 指令在所有版本的 ARM 体系结构中都有效。

`MCR2` ARM 指令可用于 ARMv5 及更高版本。

`MCRR` ARM 指令可用于 ARMv6 及更高版本，以及 ARMv5 的 E 变体。

`MCRR2` ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

4.10.3 MRC、MRC2、MRRC 和 MRRC2

从协处理器移到 ARM 寄存器。

对于不同的协处理器，有多种附加操作可供您选用。

语法

op{*cond*} *coproc*, #*opcode1*, *Rt*, *CRn*, *CRm*{, #*opcode2*}

op{*cond*} *coproc*, #*opcode1*, *Rt*, *Rt2*, *CRm*

其中：

op 是 MRC、MRC2、MRRC 或 MRRC2。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。在 ARM 代码中，MRC2 或 MRRC2 不允许使用 *cond*。

coproc 是要运行指令的协处理器的名称。标准名称为 p*n*，其中 *n* 为 0 到 15 范围内的整数。

opcode1 是一个协处理器特定的操作码。

Rt, *Rt2* 是 ARM 源寄存器。不要使用 r15。*Rt2* 仅适用于 MRRC 和 MRRC2。

在 MRC 和 MRC2 中，*Rt* 可为 APSR_nzcv。

CRn, *CRm* 是协处理器寄存器。*CRn* 仅适用于 MRC 和 MRC2。

opcode2 是一个依协处理器而定的可选操作码，它适用于 MRC 和 MRC2。

用法

这些指令的具体用法取决于协处理器。有关详细信息，请参阅协处理器文档。

体系结构

`MRC` ARM 指令在所有版本的 ARM 体系结构中都有效。

`MRC2` ARM 指令可用于 ARMv5 及更高版本。

`MRRC` ARM 指令可用于 ARMv6 和更高版本，以及 ARMv5 的 E 变体。

`MRRC2` ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

4.10.4 LDC、LDC2、STC 和 STC2

在内存和协处理器之间传送数据。

语法

op{L}{*cond*} *coproc*, *CRd*, [*Rn*]

op{L}{*cond*} *coproc*, *CRd*, [*Rn*, #{-}*offset*]{!}

op{L}{*cond*} *coproc*, *CRd*, [*Rn*], #{-}*offset*

op{L}{*cond*} *coproc*, *CRd*, *label*

其中:

op 是 LDC、LDC2、STC 或 STC2 之一。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

在 ARM 代码中，LDC2 或 STC2 不允许使用 *cond*。

L 是一个可选的后缀，它指定一个长整数传送。

coproc 是要运行指令的协处理器的名称。标准名称为 p*n*，其中 *n* 为 0 到 15 范围内的整数。

CRd 是要加载或存储的协处理器寄存器。

Rn 是内存地址所基于的寄存器。如果指定了 r15，则使用的值是当前指令地址加 8。

\- 是一个可选的减号。如果存在 -，则会从 *Rn* 中减去偏移量。否则，会将偏移量加到 *Rn* 中。

offset 是一个表达式，其值为 0 到 1020 范围内的 4 的倍数。

! 是一个可选的后缀。如果有 !，则包含偏移量的地址将被写回到 *Rn* 中。

label 是一个相对于程序的字对齐表达式。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

label 必须位于当前指令的 1020 字节范围内。

用法

这些指令的具体用法取决于协处理器。有关详细信息，请参阅协处理器文档。

在 Thumb-2EE 中，如果基址寄存器中的值为零，则代码将跳转到位于 HandlerBase - 4 的 NullCheck 处理程序处执行。

体系结构

`LDC` 和 `STC` 在所有版本的 ARM 体系结构中都有效。

`LDC2` 和 `STC2` 可用于 ARMv5 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些指令均无 16 位 Thumb 版本。

注释

对于 ARMv6T2 及更高版本，请不要在 `STC` 和 `STC2` 指令中使用 PC 相对的寻址。

4.11 其他指令

本节包括以下小节:

- 第4-129 页的*BKPT*

  断点。

- 第4-130 页的*SVC*

  超级用户调用(以前为 `SWI`)。

- 第4-131 页的*MRS*

  将 CPSR 或 SPSR 的内容移到一个通用寄存器中。

- 第4-133 页的*MSR*

  将立即数或通用寄存器的内容加载到 CPSR 或 SPSR 的指定字段中。

- 第4-135 页的*CPS*

  更改处理器状态。

- 第4-137 页的*SMC*

  安全监控调用(以前为 `SMI`)。

- 第4-138 页的*SETEND*

  设置 CPSR 中的端序位。

- 第4-139 页的*NOP*、*SEV*、*WFE*、*WFI 和 YIELD*

  无操作、设置事件、等待事件、等待中断和通知提示指令。

- 第4-141 页的*DBG*、*DMB*、*DSB 和 ISB*

  调试指令、数据内存屏障指令、数据同步屏障指令和指令同步屏障提示指令。

- 第4-143 页的*MAR 和 MRA*

  XScale 协处理器 0 指令。

  在两个通用寄存器和一个 40 位内部累加器之间传送。

4.11.1 BKPT

断点。

语法

```
BKPT #immed
```

其中:

immed 是一个表达式，其取值为以下范围内的一个整数:

- 在 ARM 指令中，为 0-65535（16 位值）
- 在 16 位 Thumb 指令中，为 0-255（8 位值）。

用法

`BKPT` 指令可使处理器进入调试状态。当指令到达某个特定地址处时，调试工具可以使用此指令来检查系统状态。

在 ARM 和 Thumb 状态中，ARM 硬件会忽略 *immed*。不过，调试器可用它来存储断点的其他信息。

体系结构

此 ARM 指令可用于 ARMv5 及更高版本。

此 16 位 Thumb 指令可用于 ARMv5 及更高版本的 T 变体。

此指令无 32 位 Thumb 版本。

4.11.2 SVC

超级用户调用。

语法

```
SVC{cond} #immed
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

immed 是一个表达式，其取值为以下范围内的一个整数:

- 在 ARM 指令中，为 0 到 2^{24}-1（24 位值）
- 在 16 位 Thumb 指令中，为 0-255（8 位值）。

用法

SVC 指令会引发一个异常。这意味着处理器模式将改变为超级用户模式，CPSR 将会被保存到超级用户模式 SPSR，并且代码将跳转到 SVC 向量执行（请参阅 RealView 编译工具开发指南 中的第 6 章 *处理处理器异常*）。

处理器会忽略 *immed*。但异常处理程序会获取它，借以确定所请求服务。

——注意——

作为 ARM 汇编语言开发成果的一部分，SWI 指令已被重命名为 SVC。在此版本的 RVCT 中，SWI 指令反汇编为了 SVC，配有注释以指明为以前的 SWI。

条件标记

此指令不更改标记。

体系结构

此 ARM 指令可用于所有版本的 ARM 体系结构。

此 16 位 Thumb 指令可用于 ARM 体系结构的所有 T 变体，以及 ARMv6 和更高版本的 T2 变体。

4.11.3 MRS

将 PSR 的内容移到通用寄存器中。

语法

```
MRS{cond} Rd, psr
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是目标寄存器。*Rd* 不能为 r15。

psr 是下列项之一:

APSR	任何处理器，任何模式。
CPSR	推荐使用的 APSR 的同义词，用于调试状态。
SPSR	除 ARMv7-M 外的任何处理器，仅可用于特权模式。
Mpsr	仅用于 ARMv7-M 处理器。

Mpsr 是下列项之一: IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、DSP、PRIMASK、BASEPRI、BASEPRI_MAX 或 CONTROL

用法

可将 MRS 与 MSR 结合使用，创建一个更新 PSR 的读- 改- 写序列，例如更改处理器模式或清除 Q 标记。

在进程交换代码中，必须保存程序员的换出进程的模型状态，包括 PSR 的相关内容。同样，也必须恢复换入进程的状态。这些操作使用的是 MRS/存储和加载/MSR 指令序列。

SPSR

当处理器处于用户或系统模式时，请不要访问 SPSR。这是您的责任。汇编程序无法就此发出警告，因为它不知道执行过程中的处理器模式。

如果在处理器处于用户模式或系统模式时，尝试访问 SPSR，则结果将是无法预料的。

CPSR

仅当处理器处于调试状态、暂停调试模式时，才能读取 CPSR 执行状态位。否则，CPSR 中的执行状态位的读取结果将会为零。

条件标记的读取不受模式和处理器的限制。应使用 APSR，而不是 CPSR。

条件标记

此指令不更改标记。

体系结构

此 ARM 指令可用于所有版本的 ARM 体系结构。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

此指令无 16 位 Thumb 版本。

4.11.4 MSR

将通用寄存器的立即数或内容加载*程序状态寄存器* (PSR) 的指定位段中。

语法（ARMv7-M 除外）

```
MSR{cond} APSR_flags, #constant
MSR{cond} APSR_flags, Rm
MSR{cond} psr_fields, #constant
MSR{cond} psr_fields, Rm
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

flags 指定要移动的 APSR 标记。*flags* 可以是以下的一个或多个指令:

- nzcvq ALU 可标记位段掩码，PSR[31:27]（用户模式）
- g SIMD GE 可标记位段掩码，PSR[19:16]（用户模式）。

constant 是取值为常数的一个表达式。该常数必须对应于一个 8 位结构，可通过在 32 位字内循环移动偶数位而得到。在 Thumb 中不可用。

Rm 是源寄存器。

psr 是下列项之一:

- CPSR 用于调试状态，请不要将其看做 APSR 的同义词
- SPSR 仅可用于特权模式下的处理器。

fields 指定要移动的 SPSR 或 CPSR 位段。*fields* 可以是以下一个或多个值:

- c 控制位段掩码字节，PSR[7:0]（特权模式）
- x 扩展位段掩码字节，PSR[15:8]（特权模式）
- s 状态位段掩码字节，PSR[23:16]（特权模式）
- f 标记位段掩码字节，PSR[31:24]（特权模式）。

语法 (ARMv7-M)

```
MSR{cond} psr, Rm
```

其中:

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。

Rm 是源寄存器。

psr 是下列项之一:APSR、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、DSP、PRIMASK、BASEPRI、BASEPRI_MAX 或 CONTROL。仅限 ARMv7-M。

用法

请参阅第4-131 页的*MRS*。

在用户模式中:

- 使用 APSR 访问条件标记、Q 或 GE 位。
- 忽略对 CPSR 中的未分配状态位、特权状态位或执行状态位的写入。这可确保用户模式程序不会更改为 特权模式。

如果在用户或系统模式时尝试访问 SPSR,则结果将是无法预料的。

条件标记

如果指定了 APSR_nzcvq 或 CPSR_f 位段,则此指令将会显式更新标记。

体系结构

此 ARM 指令可用于所有版本的 ARM 体系结构。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

此指令无 16 位 Thumb 版本。

4.11.5 CPS

`CPS`（更改处理器状态）会更改 CPSR 中的一个或多个模式以及 A、I 和 F 位，但不更改其他 CPSR 位。

`CPS` 只允许在特权模式下使用，在用户模式下不起作用。

`CPS` 不能是有条件的并且不允许在 IT 块中使用。

语法

`CPS`*`effect iflags`*`{, #`*`mode`*`}`

`CPS #`*`mode`*

其中：

effect 是下列项之一：

- `IE` 启用中断或中止。
- `ID` 禁用中断或中止。

iflags 是由下面一个或多个指令组成的序列。

- `a` 启用或禁用不精确的中止。
- `i` 启用或禁用 IRQ 中断。
- `f` 启用或禁用 FIQ 中断。

mode 指定要更改到的目标模式的编号。

条件标记

此指令不更改条件标记。

16 位指令

这些指令的下列格式可用于 Thumb 代码中，在 Thumb-2 代码中使用时为 16 位指令：

`CPSIE` *`iflags`* 您不能在 16 位 Thumb 指令中指定模式更改。

`CPSID` *`iflags`* 您不能在 16 位 Thumb 指令中指定模式更改。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

此 16 位 Thumb 指令可用于 ARMv6 及更高版本的 T 变体。

示例

```
CPSIE if      ; enable interrupts and fast interrupts
CPSID A       ; disable imprecise aborts
CPSID ai, #17 ; disable imprecise aborts and interrupts, and enter FIQ mode
CPS #16       ; enter User mode
```

4.11.6 SMC

安全监控调用。

有关详细信息，请参阅 *《ARM 体系结构参考手册安全扩展补充》*。

语法

SMC{*cond*} #*immed_16*

其中：

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

immed_16 是一个 16 位立即值。ARM 处理器会忽略此值，但 SMC 异常处理程序可以使用它来确定所请求的服务。

注释

作为 ARM 汇编语言开发成果的一部分，SMI 指令已被重命名为 SMC。在此版本的 RVCT 中，SMI 指令被反汇编为 SMC，并提供注释以指明这是以前的 SMI。

体系结构

如果 ARMv6 及更高版本的处理器具有安全扩展，则此 ARM 指令就可在这些处理器上实现。

如果 ARMv6T2 和 ARMv7 具有安全扩展，则此 32 位 Thumb 指令就可在这些处理器上实现。

此指令无 16 位 Thumb 版本。

4.11.7 SETEND

设置 CPSR 中的端序位，不影响 CPSR 中的其他位。

SETEND 不可是有条件的，并且不允许在 IT 块中使用。

语法

SETEND *specifier*

其中：

specifier 是下列项之一：

- BE 大端。
- LE 小端。

用法

使用 SETEND 可以访问采用不同端序的数据，例如通过其他采用小端存储方式的应用程序，来访问多个采用大端存储方式的 DMA 格式数据字段。

体系结构

此 ARM 指令可用于 ARMv6 及更高版本。

此 16 位 Thumb 指令可用于 ARMv6 及更高版本的 T 变体，ARMv7-M 架构除外。

此指令无 32 位 Thumb 版本。

示例

```
SETEND  BE          ; Set the CPSR E bit for big-endian accesses
LDR     r0, [r2, #header]
LDR     r1, [r2, #CRC32]
SETEND  le          ; Set the CPSR E bit for little-endian accesses for the
                    ; rest of the application
```

4.11.8 NOP、SEV、WFE、WFI 和 YIELD

无操作、设置事件、等待事件、等待中断和通知。

语法

```
NOP{cond}
```

```
SEV{cond}
```

```
WFE{cond}
```

```
WFI{cond}
```

```
YIELD{cond}
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

用法

这些是提示指令。是否实现这些指令是可选的。只要其中任何一个指令未实现，则与 `NOP` 效果相同。

NOP

`NOP` 不执行任何操作。在您的目标体系结构中，如果 `NOP` 不作为特殊指令来实现，则汇编程序会生成不执行任何操作的另一条指令，例如 `MOV r0, r0` (ARM) 或 `MOV r8, r8` (Thumb)。

`NOP` 并非一定就是消耗时间的 `NOP`。也许在该指令执行前，处理器就会将其从管道中删除。

您可利用 `NOP` 来完成填充，例如将以下指令置于 64 位边界上。

SEV

`SEV` 会导致向多处理器系统的所有内核发送事件信息。如果实现了 `SEV`，则还必须实现 `WFE`。

WFE

如果未设置事件寄存器，则 `WFE` 会暂时中断挂起执行，直至发生任一以下事件后再恢复执行：

- 发生 IRQ 中断，除非被 CPSR I 位屏蔽
- 发生 FIQ 中断，除非被 CPSR F 位屏蔽
- 发生不精确的数据中止，除非被 CPSR A 位屏蔽
- 出现调试进入请求（需启用调试）
- 另一个处理器利用 `SEV` 指令向事件发送信号。

如果设置了事件寄存器，则 `WFE` 会清除该设置，然后立即返回。

如果实现了 `WFE`，则还必须实现 `SEV`。

WFI

`WFI` 会暂时将执行中断挂起，直至发生以下事件后再恢复执行：

- 发生 IRQ 中断，不考虑 CPSR I 位
- 发生 FIQ 中断，不考虑 CPSR F 位
- 发生不精确的数据中止，除非被 CPSR A 位屏蔽
- 出现调试进入请求，无论是否启用调试。

YIELD

`YIELD` 可告知硬件有线程正在执行任务，例如可换出的自旋锁。硬件可使用此提示在多线程系统中暂挂并恢复执行线程。

体系结构

这些 ARM 指令可用在 ARMv6T2 和 ARMv7 中。

这些 32 位 Thumb 指令可用于 ARMv6T2 和 ARMv7。

这些 16 位 Thumb 指令可用于 ARMv6T2 及更高版本，以及 ARMv6 的 K 变体。

`NOP` 可在其他所有 ARM 和 Thumb 体系结构中用于伪指令。

4.11.9 DBG、DMB、DSB 和 ISB

调试指令、数据内存屏障指令、数据同步屏障指令和指令同步屏障指令。

语法

```
DBG{cond} {#option}
```

```
DMB{cond} {option}
```

```
DSB{cond} {option}
```

```
ISB{cond} {option}
```

其中:

cond	是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。
option	对提示操作的可选限制。

用法

这些是提示指令。是否实现这些指令是可选的。只要其中任何一个指令未实现，则与 NOP 效果相同。

DBG

调试提示可向调试系统及其相关系统发送提示。有关这些系统如何使用此指令的信息，请参这些系统的文档。

DMB

数据内存屏障可作为内存屏障使用。它可确保会先检测到程序中位于 DMB 指令前的所有显式内存访问指令，然后再检测到程序中位于 DMB 指令后的显式内存访问指令。它不影响其他指令在处理器上的执行顺序。

option 的允许值为:

SY	完整的系统 DMB 操作。这是缺省情况，可以省略。

DSB

数据同步屏障是一种特殊类型的内存屏障。只有当此指令执行完毕后，才会执行程序中位于此指令后的指令。当满足以下条件时，此指令才会完成：

- 位于此指令前的所有显式内存访问均完成。
- 位于此指令前的所有缓存、跳转预测和 TLB 维护操作全部完成。

允许的值为：

`SY`	完整的系统 DSB 操作。这是缺省情况，可以省略。
`UN`	只可完成于统一点的 DSB 操作。
`ST`	存储完成后才可执行的 DSB 操作。
`UNST`	只有当存储完成后才可执行的 DSB 操作，并且只会完成于统一点。

ISB

指令同步屏障可刷新处理器中的管道，因此可确保在 `ISB` 指令完成后，才从高速缓存或内存中提取位于该指令后的其他所有指令。这可确保提取时间晚于 `ISB` 指令的指令能够检测到 `ISB` 指令执行前就已经执行的上下文更改操作的执行效果，例如更改 ASID 或已完成的 TLB 维护操作，跳转预测维护操作以及对 CP15 寄存器所做的所有更改。

此外，`ISB` 指令可确保程序中位于其后的所有跳转指令总会被写入跳转预测逻辑，其写入上下文可确保 `ISB` 指令后的指令均可检测到这些跳转指令。这是指令流能够正确执行的前提条件。

`option` 的允许值为：

`SY`	完整的系统 DMB 操作。这是缺省情况，可以省略。

体系结构

这些 ARM 和 32 位 Thumb 指令可用于 ARMv7。

这些指令均无 16 位 Thumb 版本。

4.11.10 MAR 和 MRA

XScale 协处理器 0 指令。

在两个通用寄存器和一个 40 位内部累加器之间传送。

语法

MAR{*cond*} *Acc*, *RdLo*, *RdHi*

MRA{*cond*} *RdLo*, *RdHi*, *Acc*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Acc 是内部累加器。此累加器的标准名称为 acc*x*，其中 *x* 为一个整数，范围为 0 到 *n*。具体的 *n* 值取决于处理器。对于当前处理器，其值为 0。

RdLo, *RdHi* 是通用寄存器。*RdLo* 和 *RdHi* 不能为 pc，对于 MRA，它们必须为不同的寄存器。

用法

MAR 指令可将 *RdLo* 中的值复制到 *Acc* 的位 [31:0] 中，还会将 *RdHi* 的最低有效字节复制到 *Acc* 的位 [39:32] 中。

MRA 指令可进行以下操作:

- 将 *Acc* 的 [31:0] 位复制到 *RdLo*
- 将 *Acc* 的 [39:32] 位复制到 *RdHi* [7:0] 位
- 通过将 *Acc* 的 [39] 位复制到 *RdHi* 的 [31:8] 位来对该值进行符号扩展。

体系结构

这些 ARM 指令只可用于 XScale 处理器。

这些指令没有 Thumb 版本。

示例

```
MAR     acc0, r0, r1
MRA     r4, r5, acc0
MARNE   acc0, r9, r2
MRAGT   r4, r8, acc0
```


4.12 ThumbEE 指令

仅当通过使用 `--thumbx` 命令行选项或 `THUMBX` 指令将汇编程序切换为 ThumbEE 状态后，才可使用除 `ENTERX` 和 `LEAVEX` 外的其他 ThumbEE 指令。

本节包括以下小节：

- 第4-145 页的*ENTERX 和 LEAVEX*

  在 Thumb 状态和 ThumbEE 状态间切换。

- 第4-146 页的*CHKA*

  检查数组。

- 第4-147 页的*HB*、*HBL*、*HBLP 和 HBP*

  处理程序跳转，跳转到指定处理程序。

4.12.1 ENTERX 和 LEAVEX

在 Thumb 状态和 ThumbEE 状态间切换。

语法

```
ENTERX
```

```
LEAVEX
```

用法

`ENTERX` 可将 Thumb 状态更改为 ThumbEE 状态，但对 ThumbEE 状态则不起作用。

`LEAVEX` 可将 ThumbEE 状态更改为 Thumb 状态，但对 Thumb 状态则不起作用。

请不要在 IT 块中使用 `ENTERX` 或 `LEAVEX`。

体系结构

ARM 指令集中没有这些指令。

这些 32 位 Thumb 和 Thumb 2EE 指令可在 Thumb 2EE 的支持下用于 ARMv7 中。

这些指令没有 16 位 Thumb 2 版本。

有关详细信息，请参阅 *《ARM 体系结构参考手册 Thumb-2 执行环境补充》*。

4.12.2 CHKA

CHKA（检查数组）可比较两个寄存器中的无符号值。

如果第一个寄存器中的值小于或等于第二个寄存器中的值，则会将 pc 复制到 lr，并跳转到 IndexCheck 处理程序。

语法

CHKA *Rn*, *Rm*

其中:

Rn 存放数组的大小值。可为 r0-r12、SP 或 LR 中的任意一个。

Rm 存放数组索引。可为 r0-r12 或 LR 中的任意一个。

体系结构

此指令在 ARM 状态下无效。

此 16 位 ThumbEE 指令需要 Thumb 2EE 的支持。且仅可用于 ARMv7 中。

4.12.3 HB、HBL、HBLP 和 HBP

处理程序跳转，跳转到指定处理程序。

此指令可将返回地址存储到 lr，也可将参数传递到处理程序，同时还可两个操作都执行，具体您可进行选择。

语法

HB{L} *#HandlerID*

HB{L}P *#immed*, *#HandlerID*

其中:

L 是一个可选的后缀。如果选择 L，则指令会将返回地址保存到 lr 中。

P 是一个可选的后缀。如果选择 P，则指令会将 *immed* 值传递给 r8 的处理程序。

immed 是一个立即数。如果选择 L，则 *immed* 必须在范围 0-31 内，否则 *immed* 必须在范围 0-7 内。

HandlerID 是要调用的处理程序的索引编号。如果选择 *P*，则 *HandlerID* 必须在 0-31 范围内，否则 *HandlerID* 必须在 0-255 范围内。

体系结构

这些指令在 ARM 状态下无效。

这些 16 位 ThumbEE 指令需要 Thumb-2EE 的支持，且仅可用于 ThumbEE 状态下的 ARMv7。

4.13 伪指令

ARM 汇编程序支持诸多伪指令，这些伪指令在汇编时将会被解释为相应的 ARM、Thumb 或 Thumb-2 指令的组合。

下面对这些伪指令进行介绍：

- 第4-149 页的*ADRL 伪指令*

  将相对于程序或相对于寄存器的地址载入寄存器中（中等范围，与位置无关）

- 第4-151 页的*MOV32 伪指令*

  将 32 位常数或地址载入寄存器（无范围限制，但与位置相关）。仅可用于 ARMv6T2 及更高版本。

- 第4-153 页的*LDR 伪指令*

  将 32 位常数或地址载入寄存器（无范围限制，但与位置相关）。可用于所有 ARM 体系结构。

- 第4-156 页的*UND 伪指令*

  生成无体系结构定义的指令。可用于所有 ARM 体系结构。

4.13.1 ADRL 伪指令

将相对于程序或相对于寄存器的地址载入寄存器中。与 ADR 指令相似。ADRL 所加载的地址比 ADR 所加载的地址更宽，因为它可生成两个数据处理指令。

——注意——

汇编版本老于 ARMv6T2 的处理器的 Thumb 指令时，ADRL 是无效的。

语法

`ADRL{cond} Rd,label`

其中：

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是要加载的寄存器。

label 表达式，与程序或寄存器相关。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

用法

ADRL 始终汇编为两个 32 位指令。即使使用单个指令就可完成地址访问，也会生成多余的第二个地址。

如果汇编程序无法将地址构建为两个指令，则它将生成一条错误消息，汇编将失败。有关加载更宽范围地址的信息，请参阅第4-153 页的*LDR 伪指令*（另请参阅第2-25 页的*将常数加载到寄存器*）。

ADRL 可生成与位置无关的代码，因为地址与程序或寄存器有关。

如果 *label* 与程序有关，则其表示的地址必须要与 ADRL 伪指令在同一汇编程序区域内，请参阅第7-65 页的*AREA*。

如果使用 ADRL 来为 BX 或 BLX 指令生成目标，则当目标中包含 Thumb 指令时，您就要自己设置地址的 Thumb 位（位 0）。

体系结构和范围

可用范围取决于所用的指令集：

ARM ±64KB 到字节或半字对齐的地址。

±256KB 字节，字对齐地址。

32 位 Thumb ±1MB 字节，字节、半字或字对齐地址。

16 位 Thumb `ADRL` 不可用。

上面给出的范围是相对于位于当前指令后的、离当前指令有四个字节（在 Thumb 代码中）或两个字（在 ARM 代码中）间隔的点而言的。在 ARM 和 32 位 Thumb 中，如果地址为 16 字节对齐，或与该点的相对性更高，则相对地址的范围可更大。

4.13.2 MOV32 伪指令

将以下项之一加载到寄存器:

- 一个 32 位常数值
- 任何地址。

`MOV32` 始终会生成两个 32 位指令，即一个 `MOV`、`MOVT` 对。您可利用它加载任何 32 位常数或访问整个地址空间。

如果用 `MOV32` 加载地址，则所生成的代码将与位置有关。

语法

`MOV32{`*`cond`*`}` *`Rd`*`,` *`expr`*

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

Rd 是要加载数据的寄存器。*Rd* 不可为 sp 或 pc。

expr 可以是下列项之一：

symbol	程序区域中的标签。
constant	任何 32 位常数。
symbol + constant	标签加上 32 位常数。

用法

MOV32 伪指令的主要功能有:

- 当单个指令中无法生成立即数时，生成文字常数。
- 将相对于程序的地址或外部地址载入寄存器中。无论链接器将包含 MOV32 的 ELF 代码段置于何处，该地址始终有效。

———注意———

以这种方式加载的地址是在链接时确定的，因此代码*不是*位置无关的。

———————————

如果所引用的标签位于 Thumb 代码中，则 MOV32 将会设置该地址的 Thumb 位（位 0）。

体系结构

此伪指令在 ARMv6T2 和 ARMv7 中的 ARM 和 Thumb 状态下均有效。

4.13.3 LDR 伪指令

将以下项之一载入寄存器:

- 一个 32 位常数值
- 一个地址。

———注意———

本节仅介绍 LDR *伪* 指令。有关 LDR *指令* 的详细信息，请参阅 第4-11 页的*内存访问指令*。

有关使用 LDR 伪指令加载常数的信息，请参阅第2-30 页的*用 LDR Rd, =const 加载*。

———————————

语法

```
LDR{cond}{.w} Rt,=[expr | label-expr]
```

其中:

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

.W 是可选的指令宽度说明符。

Rt 是要加载的寄存器。

expr 取值为一个数值常数:

- 如果 *expr* 的值位于范围内，则汇编程序将会生成一个 MOV 或 MVN 指令。
- 如果 *expr* 的值*不在* MOV 或 MVN 指令的范围内，则汇编程序会将常数放入文字池中，并会生成一个相对于程序的 LDR 指令，该指令可从文字池中读取此常数。

label-expr 是一个与程序相关的表达式或外部表达式。汇编程序会将 *label-expr* 的值放入文字池中，并会生成一个与程序有关的 LDR 指令，该指令可从文字池中加载该值。

如果 *label-expr* 是一个外部表达式，或未包含在当前代码段内，则汇编程序将会在目标文件中放入一个链接器重新定位指令。链接器将在链接时生成该地址。

如果 *label-expr* 是一个局部标签（请参阅第3-26 页的*局部标签*），则汇编程序会在目标文件中放入一个链接器重新定位指令，并会为该局部标签生成一个符号。该地址将在链接时生成。如果局部标签引用了 Thumb 代码，则还会设置该地址的 Thumb 位（位 0）。

——注意——

在 RVCT2.2 中，没有对地址的 Thumb 位进行设置。如果此设置会影响您的代码，则请使用命令行选项 `--untyped_local_labels` 迫使汇编程序在引用 Thumb 代码中的标签时不设置 Thumb 位。

用法

`LDR` 伪指令的主要功能如下:

- 当立即数由于超出了 `MOV` 和 `MVN` 指令的范围，而不能被移入寄存器中时，生成文字常数。
- 将相对于程序的地址或外部地址载入寄存器中。无论链接器将包含 `LDR` 的 ELF 代码段置于何处，该地址始终有效。

  ——注意——

  以这种方式加载的地址是在链接时确定的，因此代码*不是*位置无关的。

  ---

pc 到文字池中的值的偏移量必须小于 ±4KB（ARM、32 位 Thumb-2），或在 0 到 +1KB（Thumb、16 位 Thumb-2）范围内。您必须确保有一个满足范围要求的文字池。有关详细信息，请参阅第7-18 页的*LTORG*。

如果所引用标签在 Thumb 代码中，`LDR` 伪指令将会设置 *`label-expr`* 的 Thumb 位（位 0）。

有关如何使用 `LDR` 的详细说明，以及有关 `MOV` 和 `MVN` 的详细信息，请参阅第2-25 页的*将常数加载到寄存器*。

Thumb 代码中的 LDR

对于 ARMv6T2 及更高版本的 Thumb 代码，您可以使用 `.W` 宽度说明符强制 `LDR` 生成 32 位指令。`LDR.W` 始终生成 32 位指令，即使利用 16 位 `MOV` 就可完成常数的加载，或在 16 位 pc 相对载入范围内有文字池。

如果在第一次汇编时，汇编程序尚不知道常数值的相关信息，则不带 `.W` 的 `LDR` 将会在 Thumb 代码中生成 16 位指令，即使这会导致对于可在 32 位 `MOV` 或 `MVN` 指令中生成的常数，会通过 16 位 pc 相对加载来完成其加载。但是，如果在第一次汇编时汇编程序就已经知道了该常数，并且该常数可以通过 32 位 `MOV` 或 `MVN` 指令生成，则将会使用 `MOV` 或 `MVN` 指令。

`LDR` 伪指令不会生成 16 位标记设置 `MOV` 指令。可使用 `--diag_warning 1727` 汇编程序命令行选项来检查是否使用了 16 位指令。

有关如何在不利用文字池加载的情况下来生成常数或地址的信息，请参阅第4-151 页的*MOV32 伪指令*。

示例

```
        LDR     r3,=0xff0       ; loads 0xff0 into r3
                                ; =>  MOV.W r3,#0xff0
        LDR     r1,=0xfff       ; loads 0xfff into r1
                                ; =>  LDR r1,[pc,offset_to_litpool]
                                ;     ...
                                ;     litpool DCD 0xfff
        LDR     r2,=place       ; loads the address of
                                ; place into r2
                                ; =>  LDR r2,[pc,offset_to_litpool]
                                ;     ...
                                ;     litpool DCD place
```


4.13.4 UND 伪指令

生成无体系结构定义的指令。执行未定义指令会引发未定义指令异常。请让无体系结构定义的指令保持在未定义状态。

语法

```
UND{cond}{.w} {#expr}
```

其中:

cond 是一个可选的条件代码(请参阅第2-17 页的*条件执行*)。对于 ARM 代码或 16 位 Thumb 代码，不允许此伪指令使用 *cond*。

`.W` 是可选的指令宽度说明符。

expr 取值为一定范围内的数值常数:

- 对于 ARM 代码，为 0-65535
- 对于 32 位 Thumb 代码，为 0-4095
- 对于 16 位 Thumb 代码，为 0-255。

如果省略了 *expr*，则使用的值为 0。

Thumb 代码中的 UND

对于 ARMv6T2 及更高版本处理器的 Thumb 代码，您可利用 `.W` 宽度说明符强制 `UND` 生成 32 位指令。`UND.W` 始终生成 32 位指令，即使 *expr* 在 0-255 范围之内。

反汇编

此伪指令生成的编码将反汇编为 `DCI`。

第 5 章
NEON 和 VFP 编程

本章提供有关用汇编语言对 NEON™ 和 VFP 协处理器进行编程的参考信息。它包含以下几节:

有关个别指令的描述的位置，请参阅表5-1、第5-5 页的表5-2 和第5-6 页的表5-3。

表5-1 NEON 指令的位置

助记符	简单说明	页码
VABA、VABD	差值绝对值，差值绝对值累加	第5-51页
VABS	绝对值	第5-52页
VACGE、VACGT	绝对值比较大于或等于，绝对值比较大于	第5-30页
VACLE、VACLT	绝对值比较小于或等于，绝对值比较小于（伪指令）	第5-80页
VADD	加法	第5-53页
VADDHN	加法，选择高半部分	第5-54页
VAND	按位与	第5-26页
VAND	按位与（伪指令）	第5-79页
VBIC	按位位清除（寄存器）	第5-26页
VBIC	按位位清除（立即数）	第5-27页
VBIF、VBIT、VBSL	为 False 时按位插入，为 True 时按位插入，按位选择	第5-28页
VCEQ、VCLE、VCLT	比较等于，比较小于或等于，比较小于	第5-31页
VCLE、VCLT	比较小于或等于，比较小于（伪指令）	第5-81页
VCLS、VCLZ、VCNT	前导符号位计数，前导零计数和设置位计数	第5-59页
VCVT	将定点数或整数转换为浮点数，将浮点数转换为整数或定点数	第5-34页
VDUP	将标量复制到向量的所有向量线	第5-35页
VEXT	提取	第5-36页
VCGE、VCGT	比较大于或等于，比较大于	第5-31页
VEOR	按位异或	第5-26页
VHADD、VHSUB	半加、半减	第5-55页
VMAX、VMIN	最大值、最小值	第5-58页

表5-1 NEON 指令的位置 (续)

助记符	简单说明	页码
VLD	向量加载	第5-68页
VMLA、VMLS	乘加，乘减（向量）	第5-64页
VMLA、VMLS	乘加，乘减（按标量）	第5-65页
VMOV	移动（立即数）	第5-37页
VMOV	移动（寄存器）	第5-29页
VMOVL、VMOV{U}N	长移，窄移（寄存器）	第5-38页
VMUL	乘法（向量）	第5-64页
VMUL	乘法（按标量）	第5-65页
VMVN	反移（立即数）	第5-37页
VNEG	求反	第5-52页
VORN	按位或非	第5-26页
VORN	按位或非（伪指令）	第5-79页
VORR	按位或（寄存器）	第5-26页
VORR	按位或（立即数）	第5-27页
VPADD、VPADAL	按对加，按对加累加	第5-56页
VPMAX、VPMIN	按对最大值，按对最小值	第5-58页
VQABS	绝对值，饱和	第5-52页
VQADD	加法，饱和	第5-53页
VQDMLAL、VQDMLSL	饱和加倍乘加以及饱和加倍乘减	第5-66页
VQMOV{U}N	饱和移动（寄存器）	第5-38页
VQDMUL	饱和加倍乘法	第5-66页
VQDMULH	返回高半部分的饱和加倍乘法	第5-67页
VQNEG	求反，饱和	第5-52页

表5-1 NEON 指令的位置 (续)

助记符	简单说明	页码
VQRDMULH	返回高半部分的饱和加倍乘法	第5-67页
VQRSHL	左移，舍入，饱和（按有符号变量）	第5-46页
VQRSHR	右移，舍入，饱和（按立即数）	第5-48页
VQSHL	左移，饱和（按立即数）	第5-45页
VQSHL	左移，饱和（按有符号变量）	第5-46页
VQSHR	右移，饱和（按立即数）	第5-48页
VQSUB	减法，饱和	第5-53页
VRADDH	加法，选择高半部分，舍入	第5-54页
VRECPE、VRECPS	近似倒数，倒数步进	第5-60页
VREV	反转元素	第5-39页
VRHADD	半加，舍入	第5-55页
VRSHR、VRSRA	右移并舍入，右移、舍入并累加（按立即数）	第5-47页
VRSUBH	减法，选择高半部分，舍入	第5-54页
VRSQRTE、VRSQRTS	近似平方根倒数，平方根倒数步进	第5-61页
VSHL	左移（按立即数）	第5-45页
VSHR	右移（按立即数）	第5-47页
VSLI	左移并插入	第5-49页
VSRA	右移，累加（按立即数）	第5-47页
VSRI	右移并插入	第5-49页
VST	向量存储	第5-68页
VSUB	减法	第5-53页
VSUBH	减法，选择高半部分	第5-54页
VSWP	交换向量	第5-40页

表5-1 NEON 指令的位置 (续)

助记符	简单说明	页码
VTBL、VTBX	向量表查找	第5-41页
VTST	测试位	第5-32页
VTRN	向量转置	第5-42页
VUZP、VZIP	向量交叉存取和反向交叉存取	第5-43页

表5-2 共享 NEON 和 VFP 指令的位置

助记符	简单说明	页码	操作数	体系结构
VLDM	加载多个	第5-20页	-	全部
VLDR	加载（另请参阅第5-77 页的*VLDR 伪指令*）	第5-19页	标量	全部
VMOV	从一个 ARM® 寄存器传送到半个双精度寄存器	第5-22页	标量	全部
	从两个 ARM 寄存器传送到双精度寄存器	第5-21页	标量	VFPv2
	从半个双精度寄存器传送到 ARM 寄存器	第5-22页	标量	全部
	从双精度寄存器传送到两个 ARM 寄存器	第5-21页	标量	VFPv2
	从单精度寄存器传送到 ARM 寄存器	第5-23页	标量	全部
	从 ARM 寄存器传送到单精度寄存器	第5-23页	标量	全部
VMRS	从 NEON 和 VFP 系统寄存器传送到 ARM 寄存器	第5-24页	-	全部
VMSR	从 ARM 寄存器传送到 NEON 和 VFP 系统寄存器	第5-24页	-	全部
VSTM	存储多个	第5-20页	-	全部
VSTR	存储	第5-19页	标量	全部

表5-3 VFP 指令的位置

助记符	简单说明	页码	操作数	体系结构
VABS	绝对值	第5-89页	向量	全部
VADD	加法	第5-90页	向量	全部
VCMP	比较	第5-92页	标量	全部
VCVT	在单精度数和双精度数之间转换	第5-93页	标量	全部
	在浮点数和整数之间转换	第5-94页	标量	全部
	在浮点数和定点数之间转换	第5-95页	标量	VFPv3
VDIV	除法	第5-90页	向量	全部
VMLA	乘加	第5-91页	向量	全部
VMLS	乘减	第5-91页	向量	全部
VMOV	将浮点常数插入单精度或双精度寄存器（另请参阅第5-5 页的表5-2）	第5-96页	标量	VFPv3
VMUL	乘法	第5-91页	向量	全部
VNEG	求反	第5-89页	向量	全部
VNMLA	求反乘加	第5-91页	向量	全部
VNMLS	求反乘减	第5-91页	向量	全部
VNMUL	求反乘法	第5-91页	向量	全部
VSQRT	平方根	第5-89页	向量	全部
VSUB	减法	第5-90页	向量	全部

5.1 扩展寄存器组

NEON 和 VFPv3 使用相同的扩展寄存器组。该寄存器组不同于 ARM 寄存器组。它是 VFPv2 寄存器组的超集。

可按照以下各节中的说明，使用三个具有显式别名的视图来引用扩展寄存器组。

第5-8 页的图5-1 显示了扩展寄存器组的三个视图，以及单字、双字及四字寄存器的重叠方式。

5.1.1 寄存器组的 NEON 视图

NEON 在查看扩展寄存器组时将其视为:

- 十六个 128 位四字寄存器 `Q0-Q15`。
- 三十二个 64 位双字寄存器 `D0-D31`。此视图也可在 VFPv3 中使用。
- 上述视图中的寄存器组合。

NEON 在查看寄存器时将每个寄存器视为均包含一个*向量*，而该向量又包含 1、2、4、8 或 16 个大小和类型均相同的元素。也可以将各元素当作*标量*加以访问。

5.1.2 扩展寄存器组的 VFPv3 视图

在 VFPv3 中，查看扩展寄存器组时可以将其视为:

- 三十二个 64 位双字寄存器 `D0-D31`。此视图也可在 NEON 中使用。
- 三十二个 32 位单字寄存器 `S0-S31`。在此视图中只能访问半个寄存器组。
- 上述视图中的寄存器组合。

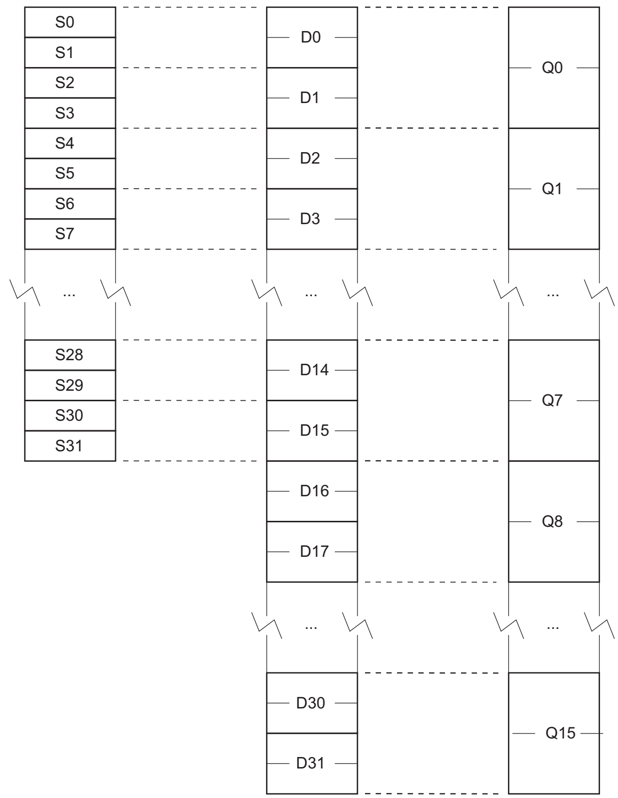


图5-1 扩展寄存器组

寄存器之间的映射如下所示:

- S<2n> 映射到 D<n> 的最低有效半部
- S<2n+1> 映射到 D<n> 的最高有效半部
- D<2n> 映射到 Q<n> 的最低有效半部
- D<2n+1> 映射到 Q<n> 的最高有效半部

例如，通过引用 D12 可以访问 Q6 中向量元素的最低有效半部，通过引用 D13 可以访问这些元素的最高有效半部。

5.2 条件代码

在 ARM 状态下，可以使用条件代码来控制 VFP 指令的执行。此类指令是根据 APSR 中的状态标记有条件执行的，执行方式与几乎所有其他 ARM 指令完全相同。

在 ARM 状态下，除 VFP 和 NEON 公用的指令之外，不能使用条件代码来控制 NEON 指令的执行。

在 Thumb-2 处理器上的 Thumb® 状态下，使用一个 `IT` 指令最多可在随后的四个 NEON 或 VFP 指令上设置条件代码。有关详细信息，请参阅第4-68 页的*IT*。

`FCMP` 是可用于更新状态标记的唯一 VFP 指令。它不是直接更新 APSR 中的标记，而是更新 FPSCR 中的一个单独的标记集（请参阅第5-82 页的*FPSCR，浮点状态和控制寄存器*）。

——注意——

若要使用这些标记来控制条件指令（包括条件 VFP 指令），必须先使用 `VMSR` 指令将其复制到 APSR（请参阅第5-24 页的*VMRS 和 VMSR*）。

对于这些标记来说，在 `FCMP` 指令后的准确含义与其在 ARM 数据处理指令后的含义是不同的。这是因为:

- 浮点值总是有符号的，因此不需要无符号的条件
- *非数字* (NaN) 值彼此之间或者与数字之间没有排序关系，因此需要附加条件来针对*无序*的结果。

条件代码助记符的含义如第5-10 页的表5-4 中所示。

表5-4 条件代码

助记符	在 ARM 数据处理指令后的含义	在 VFP FCMP 指令后的含义
EQ	等于	等于
NE	不等于	不等于或无序
CS / HS	设置进位/无符号大于或相同	大于或等于或无序
CC / LO	进位清除/无符号小于	小于
MI	负数	小于
PL	正数或零	大于或等于或无序
VS	溢出	无序（至少一个非数字操作数）
VC	无溢出	非无序
HI	无符号大于	大于或无序
LS	无符号小于或相同	小于或等于
GE	有符号大于或等于	大于或等于
LT	有符号小于	小于或无序
GT	有符号大于	大于
LE	有符号小于或等于	小于或等于或无序
AL	始终（通常省略）	始终（通常省略）

——注意——

最后更新 APSR 中的标记的指令类型确定了条件代码的含义。

5.3 一般信息

为避免重复，下面列出了许多指令共有的一些信息。本节包括以下小节:

- *浮点异常*
- *体系机构的版本*
- 第5-12 页的*NEON 和 VFP 数据类型*
- 第5-13 页的*NEON 中的正常指令、长指令、宽指令、窄指令和饱和指令*
- 第5-15 页的*NEON 标量*
- 第5-15 页的*扩展记号*
- 第5-16 页的*{0,1} 上的多项式算法*
- 第5-17 页的*VFP 协处理器*
- 第5-17 页的*VFP 寄存器*。

5.3.1 浮点异常

在会导致浮点异常的那些指令的描述中，有一小节列出了这些异常。如果某一指令描述中不包含浮点异常小节，则表明该指令不会导致任何异常。

5.3.2 体系机构的版本

所有 NEON 指令均适用于支持 NEON 的所有系统。除了在 NEON 和 VFP 之间共享的指令之外，NEON 指令只适用于支持 NEON 的系统。

在每个 VFP 和共享指令的描述中，有一小节指出了支持该指令的体系结构。

ARMv7-M 不支持 NEON 或 VFP。

5.3.3 NEON 和 VFP 数据类型

NEON 和 VFP 指令中的数据类型说明符由一个指示数据类型的字母构成，该字母通常后跟一个指示宽度的数字。在某种程度上，它们不同于指令助记符。表5-5 显示了可在 NEON 指令中使用的数据类型。表5-6 显示了可在 VFP 指令中使用的数据类型。

表5-5 NEON 数据类型

	8 位	16 位	32 位	64 位
无符号整数	U8	U16	U32	U64
有符号整数	S8	S16	S32	S64
未指定类型的整数	I8	I16	I32	I64
浮点数	不可用	不可用	F（或 F32）	不可用
{0,1} 上的多项式	P8	P16	不可用	不可用

表5-6 VFP 数据类型

	16 位	32 位	64 位
无符号整数	U16	U32	不可用
有符号整数	S16	S32	不可用
浮点数	不可用	F（或 F32）	D（或 F64）

有关 {0,1} 上多项式运算的其他信息，请参阅第5-16 页的*{0,1} 上的多项式算法*。

在指令中指定第二个（或唯一的）操作数的数据类型。

——注意——

大多数指令所允许的数据类型具有限定范围。有关详细信息，请参阅指令页。

————————

5.3.4 NEON 中的正常指令、长指令、宽指令、窄指令和饱和指令

许多 NEON 数据处理指令可具有正常指令、长指令、宽指令、窄指令和饱和指令变体形式。

NEON 指令可处理:

- 由以下内容构成的双字向量:
  - 八个 8 位元素
  - 四个 16 位元素
  - 两个 32 位元素
  - 一个 64 位元素。
- 由以下内容构成的四字向量:
  - 十六个 8 位元素
  - 八个 16 位元素
  - 四个 32 位元素
  - 两个 64 位元素。

正常指令

正常指令可对上述任意向量类型执行运算，并生成大小相同且类型通常与操作数向量相同的结果向量。

通过将 `Q` 附加到指令助记符，可以指定正常指令的操作数和结果必须全部为四字。这样指定后，如果操作数或结果不是四字，则汇编程序会生成错误。

长指令

长指令对双字向量操作数执行运算，并生成四字向量结果。所生成的元素通常是操作数元素宽度的两倍，并属于同一类型。

通过将 `L` 追加到指令助记符来指定长指令。

宽指令

宽指令对一个双字向量操作数和一个四字向量操作数执行运算。此类指令生成四字向量结果。所生成的元素和第一个操作数的元素是第二个操作数元素宽度的两倍。

通过将 `W` 追加到指令助记符来指定宽指令。

窄指令

窄指令对四字向量操作数执行运算，并生成双字向量结果。所生成的元素通常是操作数元素宽度的一半。

通过将 `N` 追加到指令助记符来指定窄指令。

饱和指令

有关饱和指令用法的一般描述，请参阅第4-93 页的*饱和指令*。有关 NEON 饱和指令饱和到的范围，请参阅表5-7。

通过在 `V` 和指令助记符之间使用 `Q` 前缀来指定饱和指令。

表5-7 NEON 饱和范围

数据类型	*x* 的饱和范围
`S8`	$-2^{7} <= x < 2^{7}$
`S16`	$-2^{15} <= x < 2^{15}$
`S32`	$-2^{31} <= x < 2^{31}$
`S64`	$-2^{63} <= x < 2^{63}$
`U8`	$0 <= x < 2^{8}$
`U16`	$0 <= x < 2^{16}$
`U32`	$0 <= x < 2^{32}$
`U64`	$0 <= x < 2^{64}$

5.3.5 NEON 标量

有些 NEON 指令可处理与向量组合使用的标量。NEON 标量可以为 8 位、16 位、32 位或 64 位。除乘法指令之外，访问标量的指令也可访问寄存器组中的任何元素。指令语法通过在双字向量中使用索引来引用标量，从而使 `Dm[x]` 表示 `Dm` 中的第 *x* 个元素。

乘法指令仅允许使用 16 位或 32 位标量，并且只能访问寄存器组中的前 32 个标量。这在乘法指令中意味着:

- 16 位标量限定为寄存器 D0-D7，其中 *x* 位于范围 0-3 内
- 32 位标量限定为寄存器 D0-D15，其中 *x* 为 0 或 1。

5.3.6 扩展记号

汇编程序向 NEON 和 VFP 体系结构汇编语法提供了一个扩展，该扩展称为*扩展记号*。利用此扩展，可以在寄存器名称中包含数据类型信息或标量索引。这样，就无需在每个指令中包含数据类型或标量索引信息。

寄存器名称可以属于以下任意类型:

无类型　　此类寄存器名称指定寄存器，但不指定寄存器所包含的数据类型，也不为寄存器中的特定标量指定任何索引。

无类型但包含标量索引

　　此类寄存器名称指定寄存器，并为寄存器中的特定标量指定索引，但不指定寄存器所包含的数据类型。

有类型　　此类寄存器名称指定寄存器以及寄存器所包含的数据类型，但不为寄存器中的特定标量指定任何索引。

有类型并包含标量索引

　　此类寄存器名称指定寄存器以及寄存器所包含的数据类型，并为寄存器中的特定标量指定索引。

使用 `SN`、`DN` 和 `QN` 指令可创建有类型寄存器和标量寄存器。有关详细信息，请参阅第7-15 页的*QN、DN 和 SN*。

5.3.7 {0,1} 上的多项式算法

使用布尔算法规则处理系数 0 和 1：

- 0 + 0 = 1 + 1 = 0
- 0 + 1 = 1 + 0 = 1
- 0 * 0 = 0 * 1 = 1 * 0 = 0
- 1 * 1 = 1。

也就是说，将两个 {0,1} 上的多项式相加与按位异或的运算相同，而将两个 {0,1} 上的多项式相乘则与整乘的运算相同，但部分积执行的是异或运算，而不是相加运算。

5.3.8 VFP 协处理器

VFP 协处理器与其关联的支持代码一起提供单精度和双精度浮点数学运算，这些运算由*《ANSI/IEEE Std. 754-1985 IEEE 二进制浮点运算标准》*定义。在本章中将该文档称为 IEEE 754 标准。有关详细信息，请参阅*RealView 编译工具库和浮点支持指南*中的第 4 章 *浮点支持*。

可以使用最多包含八个单精度数或四个双精度数的短向量，但此方法已被弃用。有关其他信息，请参阅第5-97 页的*VFP 向量模式*。

5.3.9 VFP 寄存器

VFP 协处理器具有 32 个单精度寄存器：s0 到 s31。其中每个寄存器都可以包含一个单精度浮点值或一个 32 位整数。

这 32 个寄存器还可以被视为 16 个双精度寄存器：d0 到 d15。`d`*n* 与 `s(2`*n*`)` 和 `s(2`*n*`+1)` 占用的硬件空间相同。

VFPv3 额外增加了 16 个双精度寄存器（d16 到 d31），从而扩展了 VFP 寄存器组。这些增加的寄存器与任何单精度 VFP 寄存器都不会重叠。

———注意———

如果处理器同时具有 NEON 和 VFP，则所有 NEON 寄存器都会与 VFP 寄存器重叠，详情请参阅第5-7 页的*扩展寄存器组*。

————————

您可以：

- 将一些寄存器用于单精度值，同时将一些寄存器用于双精度值，并将另一些寄存器用作 NEON 向量。
- 在不同的时间，将相同的寄存器用于单精度值、双精度值和 NEON 向量。

不要试图同时使用对应的单精度和双精度寄存器。这样做虽然不会造成损坏，但结果毫无意义。

5.4 NEON 和 VFP 共享的指令

本节包括以下小节:

- 第5-19 页的*VLDR 和 VSTR*

  扩展寄存器加载和存储。

- 第5-20 页的*VLDM*、*VSTM*、*VPOP 和 VPUSH*

  扩展寄存器加载多个和存储多个。

- 第5-21 页的*VMOV(在两个ARM 寄存器和一个扩展寄存器之间)*

  在两个 ARM 寄存器和一个 64 位扩展寄存器之间传送内容。

- 第5-22 页的*VMOV(在一个ARM 寄存器和一个NEON 标量之间)*

  在一个 ARM 寄存器和半个 64 位扩展寄存器之间传送内容。

- 第5-23 页的*VMOV(在一个ARM 寄存器和一个单精度 VFP 之间)*

  在一个 32 位扩展寄存器和一个 ARM 寄存器之间传送内容。

- 第5-24 页的*VMRS 和 VMSR*

  在一个 ARM 寄存器与一个 NEON 和 VFP 系统寄存器之间传送内容。

5.4.1 VLDR 和 VSTR

扩展寄存器加载和存储。

语法

```
VLDR{cond}{.size} Fd, [Rn{, #offset}]
VSTR{cond}{.size} Fd, [Rn{, #offset}]
VLDR{cond}{.size} Fd, label
VSTR{cond}{.size} Fd, label
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 是一个可选的数据大小说明符。如果 *Fd* 是单精度 VFP 寄存器，则必须为 32；否则必须为 64。

Fd 是要加载或保存的扩展寄存器。对于 NEON 指令，它必须为 *Dd*。对于 VFP 指令，它可以为 *Dd* 或 *Sd*。

Rn 是存放要传送的基址的 ARM 寄存器。

offset 是一个可选的数值表达式。在汇编时，该表达式的值必须为一个数字常数。该值必须是 4 的倍数，并在 -1020 到 +1020 的范围内。该值被加到基址上以构成用于传送的地址。

label 是一个程序相对的表达式。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

label 必须位于当前指令的 ±1KB 范围之内。

用法

VLDR 指令可从内存中加载一个扩展寄存器。VSTR 指令可将一个扩展寄存器的内容保存到内存中。

如果 *Fd* 是单精度寄存器（仅限 VFP），则传送一个字。否则传送两个字。

此外，还有一个 VLDR 伪指令（请参阅第5-77 页的*VLDR 伪指令*）。

5.4.2 VLDM、VSTM、VPOP 和 VPUSH

扩展寄存器加载多个、存储多个、从堆栈弹出、推入堆栈。

语法

```
VLDMmode{cond} Rn,{!} Registers
VSTMmode{cond} Rn,{!} Registers
VPOP{cond} Registers
VPUSH{cond} Registers
```

其中:

mode 必须是下列值之一：

- IA 表示在每次传送后递增地址。IA 是缺省值，可以省略。
- DB 表示在每次传送前递减地址。
- EA 表示空的升序堆栈操作。对于加载操作，该值与 DB 相同；对于保存操作，该值与 IA 相同。
- FD 表示满的降序堆栈操作。对于加载操作，该值与 IA 相同；对于保存操作，该值与 DB 相同。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Rn 是存放要传送的基址的 ARM 寄存器。

! 是可选的。! 指定必须将更新后的基址写回到 *Rn* 中。如果未指定 !，则 *mode* 必须为 IA。

Registers 是一个用大括号 { 和 } 括起的连续扩展寄存器的列表。该列表可用逗号分隔，也可以采用范围格式。列表中必须至少有一个寄存器。

可指定 S、D 或 Q 寄存器，但一定不能混用这些寄存器。D 寄存器的数目不得超过 16 个，Q 寄存器的数目不得超过 8 个。如果指定 Q 寄存器，则在反汇编时它们将显示为 D 寄存器。

注意

VPOP *Registers* 等效于 VLDM sp!,*Registers*。

VPUSH *Registers* 等效于 VSTMDB sp!,*Registers*。

可以使用上述指令的任一形式。这些指令将反汇编为 VPOP 和 VPUSH。

5.4.3 VMOV（在两个 ARM 寄存器和一个扩展寄存器之间）

在两个 ARM 寄存器与一个 64 位扩展寄存器或两个连续的 32 位 VFP 寄存器之间传送内容。

语法

VMOV{*cond*} *Dm*, *Rd*, *Rn*

VMOV{*cond*} *Rd*, *Rn*, *Dm*

VMOV{*cond*} {*Sm*, *Sm1*}, *Rd*, *Rn*

VMOV{*cond*} *Rd*, *Rn*, {*Sm*, *Sm1*}

其中:

cond	是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。
Dm	是一个 64 位扩展寄存器。
Sm	是一个 VFP 32 位寄存器。
Sm1	是 *Sm* 之后的下一个 VFP 32 位寄存器。
Rd、*Rn*	是 ARM 寄存器。不要使用 r15。

用法

VMOV *Dm*, *Rd*, *Rn* 将 *Rd* 的内容传送到 *Dm* 的低半部分，并将 *Rn* 的内容传送到 *Dm* 的高半部分。

VMOV *Rd*, *Rn*, *Dm* 将 *Dm* 的低半部分的内容传送到 *Rd*，并将 *Dm* 的高半部分的内容传送到 *Rn*。

VMOV *Rd*, *Rn*, {*Sm*, *Sm1*} 将 *Sm* 的内容传送到 *Rd*，并将 *Sm1* 的内容传送到 *Rn*。

VMOV {*Sm*, *Sm1*}, *Rd*, *Rn* 将 *Rd* 的内容传送到 *Sm*，并将 *Rn* 的内容传送到 *Sm1*。

体系结构

64 位指令适用于 NEON 和 VFPv2 及更高版本。

2 x 32 位指令适用于 VFPv2 及更高版本。

5.4.4 VMOV（在一个 ARM 寄存器和一个 NEON 标量之间）

在一个 ARM 寄存器和一个 NEON 标量之间传送内容。

语法

VMOV{*cond*}{.*size*} *Dn*[*x*], *Rd*

VMOV{*cond*}{.*datatype*} *Rd*, *Dn*[*x*]

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 是数据大小。可以为 8、16 或 32。如果省略，则 *size* 为 32。

datatype 是数据类型。可以为 U8、S8、U16、S16 或 32。如果省略，则 *datatype* 为 32。

Dn[*x*] 是 NEON 标量（请参阅第5-15 页的*NEON 标量*）。

Rd 是 ARM 寄存器。*Rd* 不得为 R15。

用法

VMOV *Rd*, *Dn*[x] 将 *Dn*[x] 的内容传送到 *Rd* 的最低有效字节、半字或字。

VMOV *Dn*[x], *Rd* 将 *Rd* 的最低有效字节、半字或字的内容传送到 *Sn*。

5.4.5 VMOV（在一个 ARM 寄存器和一个单精度 VFP 之间）

在一个单精度浮点寄存器和一个 ARM 寄存器之间传送内容。

语法

`VMOV{`*`cond`*`} `*`Rd`*`，`*`Sn`*

`VMOV{`*`cond`*`} `*`Sn`*`，`*`Rd`*

其中：

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Sn 是 VFP 单精度寄存器。

Rd 是 ARM 寄存器。*Rd* 不得为 R15。

用法

`VMOV` *Rd*，*Sn* 将 *Sn* 的内容传送到 *Rd*。

`VMOV` *Sn*，*Rd* 将 *Rd* 的内容传送到 *Sn*。

5.4.6 VMRS 和 VMSR

在一个 ARM 寄存器与一个 NEON 和 VFP 系统寄存器之间传送内容。

语法

VMRS{*cond*} *Rd*, *extsysreg*

VMSR{*cond*} *extsysreg*, *Rd*

其中:

cond 是一个可选的条件代码(请参阅第5-9 页的*条件代码*)。

extsysreg 是 NEON 和 VFP 系统寄存器，通常是 FPSCR、FPSID 或 FPEXC(请参阅第5-82 页的*NEON 和 VFP 系统寄存器*)。

Rd 是 ARM 寄存器。*Rd* 不得为 R15。

如果 *extsysreg* 为 FPSCR，则该 ARM 寄存器可以为 APSR_nzcv。在这种情况下，会将浮点状态标记传送到 ARM APSR 中的相应标记。

用法

VMRS 指令将 *extsysreg* 的内容传送到 *Rd*。

VMSR 指令将 *Rd* 的内容传送到 *extsysreg*。

——注意——

这些指令将使 ARM 停止运行，直到完成当前所有的 NEON 或 VFP 运算。

———————

示例

```
    VMRS    r2,FPCID
    VMRS    APSR_nzcv, FPSCR     ; transfer FP status register to ARM APSR
    VMSR    FPSCR, r4
```


5.5 NEON 逻辑运算和比较运算

本节包括以下小节:

- 第5-26 页的*VAND、VBIC、VEOR、VORN 和 VORR（寄存器）*
  按位与、位清除、异或、或非以及或（寄存器）。
- 第5-27 页的*VBIC 和 VORR（立即数）*
  按位位清除和或（立即数）。
- 第5-28 页的*VBIF、VBIT 和 VBSL*
  为 False 时按位插入，为 True 时按位插入以及按位选择。
- 第5-29 页的*VMOV、VMVN（寄存器）*
  移动和求反移动。
- 第5-30 页的*VACGE 和 VACGT*
  比较绝对值。
- 第5-31 页的*VCEQ、VCGE、VCGT、VCLE 和 VCLT*
  比较。
- 第5-32 页的*VTST*
  测试位。

5.5.1 VAND、VBIC、VEOR、VORN 和 VORR（寄存器）

VAND（按位与）、VBIC（位清除）、VEOR（按位异或）、VORN（按位或非）和 VORR（按位或）指令在两个寄存器之间执行按位逻辑运算，并将结果存放到目标寄存器中。

语法

```
Vop{cond}.{datatype} {Qd}, Qn, Qm
```

```
Vop{cond}.{datatype} {Dd}, Dn, Dm
```

其中：

op 必须是下列值之一：

- AND 逻辑“与”
- ORR 逻辑“或”
- EOR 逻辑异或
- BIC 逻辑“与”求补
- ORN 逻辑“或”求补。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 是一个可选的数据类型。汇编程序将忽略 *datatype*。

Qd、*Qn*、*Qm* 为四字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

Dd、*Dn*、*Dm* 为双字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

——注意——

对两个操作数使用同一寄存器的 VORR 是 VMOV 指令。可以用此方式使用 VORR，但反汇编结果代码时会生成 VMOV 语法。有关详细信息，请参阅第5-29 页的*VMOV、VMVN（寄存器）*。

5.5.2 VBIC 和 VORR（立即数）

VBIC（位清除（立即数））获取目标向量的每个元素，对其与一个立即数执行按位与求补运算，并将结果返回到目标向量。

VORR（按位或（立即数））获取目标向量的每个元素，对其与一个立即数执行按位或运算，并将结果返回到目标向量。

另请参阅第5-79 页的*VAND 和 VORN（立即数）*伪指令。

语法

V*op*{*cond*}.*datatype Qd*, #*imm*

V*op*{*cond*}.*datatype Dd*, #*imm*

其中:

op 必须为 BIC 或 ORR。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 I16 或 I32。

Qd 或 *Dd* 是用于存放源和结果的 NEON 寄存器。

imm 是立即数。

立即数

如果 *datatype* 为 I16，则立即数必须采用下列格式之一：

- 0x00XY
- 0xXY00。

如果 *datatype* 为 I32，则立即数必须采用下列格式之一：

- 0x000000XY
- 0x0000XY00
- 0x00XY0000
- 0xXY000000。

5.5.3 VBIF、VBIT 和 VBSL

VBIT（为 True 时按位插入）：如果第二个操作数的对应位为 1，则该指令将第一个操作数中的每一位插入目标中；否则将目标位保持不变。

VBIF（为 False 时按位插入）：如果第二个操作数的对应位为 0，则该指令将第一个操作数中的每一位插入目标中；否则将目标位保持不变。

VBSL（按位选择）：如果目标的对应位为 1，则该指令从第一个操作数中选择目标的每一位；如果目标的对应位为 0，则从第二个操作数中选择目标的每一位。

语法

```
Vop{cond}{.datatype} {Qd}, Qn, Qm
```

```
Vop{cond}{.datatype} {Dd}, Dn, Dm
```

其中:

op 必须为 BIT、BIF 或 BSL 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 是一个可选的数据类型。汇编程序将忽略 *datatype*。

Qd、*Qn*、*Qm* 为四字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

Dd、*Dn*、*Dm* 为双字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

5.5.4 VMOV、VMVN（寄存器）

向量移动（寄存器）将源寄存器中的值复制到目标寄存器中。

向量求反移动（寄存器）对源寄存器中每一位的值执行求反运算，并将结果存放到目标寄存器中。

语法

```
VMOV{cond}{.datatype} Qd, Qm
VMOV{cond}{.datatype} Dd, Qm
VMVN{cond}{.datatype} Qd, Qm
VMVN{cond}{.datatype} Dd, Qm
```

其中：

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 是一个可选的数据类型。汇编程序将忽略 `datatype`。

Qd、*Qm* 为四字运算指定目标向量和源向量。

Dd、*Dm* 为双字运算指定目标向量和源向量。

5.5.5 VACGE 和 VACGT

向量绝对值比较获取一个向量中每个元素的绝对值，并将其与另一个向量中相应元素的绝对值进行比较。如果条件为 True，则将目标向量中的相应元素全部设置为 1。否则，全部设置为 0。

另请参阅第5-80 页的*VACLE 和 VACLT*伪指令。

语法

```
VACop{cond}.F32 {Qd}, Qn, Qm
```

```
VACop{cond}.F32 {Dd}, Dn, Dm
```

其中：

op 必须是下列值之一：

- GE 绝对值大于或等于
- GT 绝对值大于。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Qd、*Qn*、*Qm* 为四字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

Dd、*Dn*、*Dm* 为双字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

结果的数据类型为 I32。

5.5.6 VCEQ、VCGE、VCGT、VCLE 和 VCLT

向量比较获取向量中每个元素的值，并将其与另一个向量中相应元素的值或零进行比较。如果条件为 True，则将目标向量中的相应元素全部设置为 1。否则，全部设置为 0。

另请参阅第5-81 页的*VCLE 和 VCLT*伪指令。

语法

VC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, #0

VC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, #0

其中：

op 必须是下列值之一：

EQ	等于
GE	大于或等于
GT	大于
LE	小于或等于（仅当第二个操作数为 #0 时）
LT	小于（仅当第二个操作数为 #0 时）。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须是下列值之一：

- 对于 EQ，为 I8、I16、I32 或 F32
- 对于 GE、GT、LE 或 LT（#0 形式除外），为 S8、S16、S32、U8、U16、U32 或 F32
- 对于 GE、GT、LE 或 LT（#0 形式），为 S8、S16、S32 或 F32。

结果的数据类型为：

- 对于操作数数据类型 I32、S32、U32 或 F32，为 I32
- 对于操作数类型 I16、S16 或 U16，为 I16
- 对于操作数数据类型 I8、S8 或 U8 ,为 I8。

Qd、*Qn*、*Qm* 为四字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

Dd、*Dn*、*Dm* 为双字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

#0 在比较时用零替换 *Qm* 或 *Dm*。

5.5.7 VTST

VTST（向量测试位）获取向量中的每个元素，并将其与另一个向量中的相应元素执行按位逻辑“与”运算。如果结果不为 0，则将目标向量中的相应元素全部设置为 1。否则，全部设置为 0。

语法

```
VTST{cond}.size {Qd}, Qn, Qm
VTST{cond}.size {Dd}, Dn, Dm
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 必须为 8、16 或 32 之一。

Qd、*Qn*、*Qm* 为四字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

Dd、*Dn*、*Dm* 为双字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

5.6 NEON 通用数据处理指令

本节包括以下小节:

- 第5-34 页的*VCVT*

  向量在定点数或整数与浮点数之间转换。
- 第5-35 页的*VDUP*

  将标量复制到向量的所有向量线。
- 第5-36 页的*VEXT*

  提取。
- 第5-37 页的*VMOV*、*VMVN(立即数)*

  移动和求反移动(立即数)。
- 第5-38 页的*VMOVL*、*V{Q}MOVN*、*VQMOVUN*

  移动(寄存器)。
- 第5-39 页的*VREV*

  反转向量中的元素。
- 第5-40 页的*VSWP*

  交换向量。
- 第5-41 页的*VTBL*、*VTBX*

  向量表查找。
- 第5-42 页的*VTRN*

  向量转置。
- 第5-43 页的*VUZP*、*VZIP*

  向量交叉存取和反向交叉存取。

5.6.1 VCVT

VCVT（向量转换）按下列方式之一转换一个向量中的每个元素，并将结果存放到另一向量中：

- 浮点数到整数
- 整数到浮点数
- 浮点数到定点数
- 定点数到浮点数。

语法

```
VCVT{cond}.type Qd, Qm {, #fbits}
VCVT{cond}.type Dd, Dm {, #fbits}
```

其中：

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

type 为向量的元素指定数据类型。必须是下列值之一：

S32.F32	浮点数到有符号整数或定点数
U32.F32	浮点数到无符号整数或定点数
F32.S32	有符号整数或定点数到浮点数
F32.U32	无符号整数或定点数到浮点数

Qd、*Qm* 为四字运算指定目标向量和操作数向量。

Dd、*Dm* 为双字运算指定目标向量和操作数向量。

fbits 如果存在，则指定定点数中的小数位数。否则，将在浮点数和整数之间转换。*fbits* 必须在范围 0 到 32 内。如果省略 *fbits*，则小数位数为 0。

舍入

整数或定点数到浮点数的转换使用向最接近的数舍入。

浮点数到整数或定点数的转换使用向零舍入。

5.6.2 VDUP

向量复制将标量复制到目标向量的每一元素。源可以是 NEON 标量或 ARM 寄存器。

语法

```
VDUP{cond}.size Qd, Dm[x]
```

```
VDUP{cond}.size Dd, Dm[x]
```

```
VDUP{cond}.size Qd, Rm
```

```
VDUP{cond}.size Dd, Rm
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 必须为 8、16 或 32。

Qd 为四字运算指定目标寄存器。

Dd 为双字运算指定目标寄存器。

Dm[*x*] 指定 NEON 标量。

Rm 指定 ARM 寄存器。*Rm* 不得为 pc。

5.6.3 VEXT

向量提取从第二个操作数向量的低位和第一个操作数的高位提取 8 位元素，将这些元素连接起来，并将结果存放到目标向量中。有关示例，请参阅图5-2。

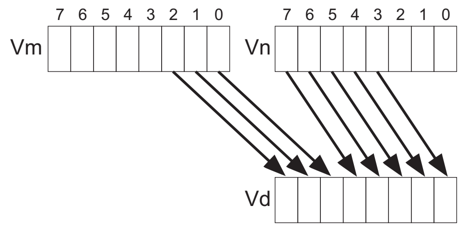


图5-2 imm 为 3 的双字 VEXT 运算

语法

```
VEXT{cond}.8 {Qd}, Qn, Qm, #imm
```

```
VEXT{cond}.8 {Dd}, Dn, Dm, #imm
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Qd、*Qn*、*Qm* 为四字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

Dd、*Dn*、*Dm* 为双字运算指定目标寄存器、第一个操作数寄存器和第二个操作数寄存器。

imm 是要从第二个操作数向量的低位提取的 8 位元素的数目；对于双字运算，位于范围 0 到 7 内；对于四字运算，位于范围 0 到 15 内。

VEXT 伪指令

可以将数据类型指定为 16、32 或 64，而不是 8。在这种情况下，`#imm` 指的是半字、字或双字而不是字节，并且允许的范围也会相应的减小。

5.6.4 VMOV、VMVN（立即数）

向量移动和向量求反移动（立即数）生成一个立即数，并将结果存放到目标寄存器。

语法

```
Vop{cond}.datatype Qd, #imm
Vop{cond}.datatype Dd, #imm
```

其中:

op 必须为 MOV 或 MVN。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 I8、I16、I32、I64 或 F32 之一。

Qd 或 *Dd* 是用于存放结果的 NEON 寄存器。

imm 是 *datatype* 所指定类型的常数。将复制此常数来填充目标寄存器。

表5-8 可用常数

datatype	VMOV	VMVN
I8	0xXY	-
I16	0x00XY、0xXY00	0xFFXY、0xXYFF
I32	0x000000XY、0x0000XY00、0x00XY0000、0xXY000000	0xFFFFFFXY、0xFFFFXYFF、0xFFXYFFFF、0xXYFFFFFF
	0x0000XYFF、0x00XYFFFF	0xFFFFXY00、0xFFXY0000
I64	字节掩码 0xGGHHJJKKLLMMNNPP [a]	-
F32	浮点数[b]	-

a. 0xGG、0xHH、0xJJ、0xKK、0xLL、0xMM、0xNN 和 0xPP 都必须为 0x00 或 0xFF。
b. 任何浮点数均可以 $+/-n * 2^{-}$ 形式表示，其中 n 和 r 是整数，16 <= n <= 31，0 <= r <= 7。

5.6.5 VMOVL、V{Q}MOVN、VQMOVUN

VMOVL（向量长移）获取双字向量中的每个元素，用符号或零将其扩展到原长度的两倍，并将结果存放到四字向量中。

VMOVN（向量窄移）将四字向量中每个元素的最低有效半部复制到双字向量的相应元素中。

VQMOVN（向量饱和窄移）将操作数向量中的每个元素复制到目标向量的相应元素中。结果元素是操作数元素宽度的一半，并且会将值饱和到结果宽度。

VQMOVUN（向量饱和窄移，有符号操作数和无符号结果）将操作数向量的每个元素复制到目标向量的相应元素中。结果元素是操作数元素宽度的一半，并且会将值饱和到结果宽度。

语法

VMOVL{*cond*}.*datatype Qd*, *Dm*

V{Q}MOVN{*cond*}.*datatype Dd*, *Qm*

VQMOVUN{*cond*}.*datatype Dd*, *Qm*

其中：

Q　如果存在，则指定对结果进行饱和。

cond　是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype　必须是下列值之一：

S8、S16、S32	对于 VMOVL
U8、U16、U62	对于 VMOVL
I16, I32, I64	对于 VMOVN
S16、S32、S64	对于 VQMOVN 或 VQMOVUN
U16、U32、U64	对于 VQMOVN。

Qd、*Dm*　为 VMOVL 指定目标向量和操作数向量。

Dd、*Qm*　为 V{Q}MOV{U}N 指定目标向量和操作数向量。

5.6.6 VREV

`VREV16`（向量在半字中反转）反转向量每个半字中的 8 位元素的顺序，并将结果存放到对应的目标向量中。

`VREV32`（向量在字中反转）反转向量每个字中的 8 位或 16 位元素的顺序，并将结果存放到对应的目标向量中。

`VREV64`（向量在双字中反转）反转向量每个双字中的 8 位、16 位或 32 位元素的顺序，并将结果存放到对应的目标向量中。

语法

VREV*n*{*cond*}.*size Qd*, *Qm*

VREV*n*{*cond*}.*size Dd*, *Dm*

其中:

n 必须为 16、32 或 64 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 必须为 8、16 或 32 之一，并且必须小于 *n*。

Qd、*Qm* 为四字运算指定目标向量和操作数向量。

Dd、*Dm* 为双字运算指定目标向量和操作数向量。

5.6.7 VSWP

VSWP（向量交换）交换两个向量的内容。这两个向量可以是双字或四字向量。它们使用相同的数据类型。

语法

```
VSWP{cond}{.datatype} Qd, Qm
```

```
VSWP{cond}{.datatype} Dd, Dm
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 是一个可选的数据类型。汇编程序将忽略 *datatype*。

Qd、*Qm* 为四字运算指定向量。

Dd、*Dm* 为双字运算指定向量。

5.6.8 VTBL、VTBX

`VTBL`（向量表查找）使用控制向量中的字节索引在表中查找字节值，并生成一个新的向量。如果索引超出范围，则返回 0。

`VTBX`（向量表扩展）的用法与上一指令相同，但索引超出范围时目标元素将保持不变。

语法

V*op*{*cond*}.8 *Dd*, *list*, *Dm*

其中：

op 必须为 `TBL` 或 `TBX`。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Dd 指定目标向量。

list 指定包含表的向量。必须是下列值之一：

- {*Dn*}
- {*Dn*,*D(n+1)*}
- {*Dn*,*D(n+1)*,*D(n+2)*}
- {*Dn*,*D(n+1)*,*D(n+2)*,*D(n+3)*}。

list 中的所有寄存器都必须位于范围 D0 到 D31 内。

Dm 指定索引向量。

5.6.9 VTRN

VTRN（向量转置）将其操作数向量的元素视为 2 x 2 矩阵的元素，并对此类矩阵进行转置。图5-3 和图5-4 显示了 VTRN 运算的示例。

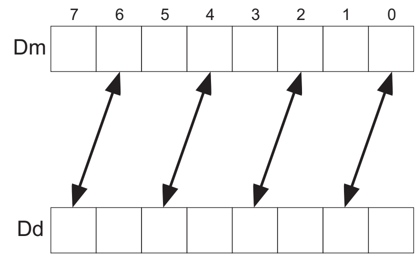


图5-3 双字 VTRN.8 的运算

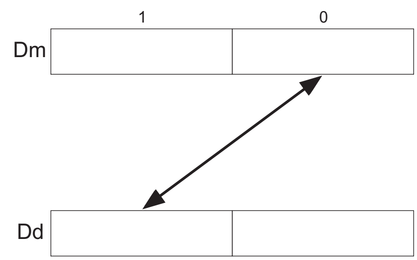


图5-4 双字 VTRN.32 的运算

语法

VTRN{*cond*}.*size Qd*, *Qm*

VTRN{*cond*}.*size Dd*, *Dm*

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 必须为 8、16 或 32 之一。

Qd、*Qm* 为四字运算指定向量。

Dd、*Dm* 为双字运算指定向量。

5.6.10 VUZP、VZIP

VZIP（向量压缩）交叉存取两个向量的元素。

VUZP（向量解压缩）反向交叉存取两个向量的元素。

有关反向交叉存取的示例，请参阅第5-68 页的*对 3 元素数组结构执行反向交叉存取*。交叉存取就是反向的过程。

语法

V*op*{*cond*}.*size Qd*, *Qm*

V*op*{*cond*}.*size Dd*, *Dm*

其中:

op 必须为 UZP 或 ZIP。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 必须为 8、16 或 32 之一。

Qd、*Qm* 为四字运算指定向量。

Dd、*Dm* 为双字运算指定向量。

注意

下列指令均是等效的:

VZIP.32 *Dd*, *Dm*

VUZP.32 *Dd*, *Dm*

VTRN.32 *Dd*, *Dm*

上述指令将反汇编为 VTRN.32 *Dd*, *Dm*。另请参阅第5-42 页的*VTRN*。

5.7 NEON 移位指令

本节包括以下小节:

- 第5-45 页的*VSHL、VQSHL、VQSHLU 和VSHLL(按立即数)*
  按立即值左移。
- 第5-46 页的*V{Q}{R}SHL(按有符号变量)*
  按有符号变量左移。
- 第5-47 页的*V{R}SHR{N}、V{R}SRA(按立即数)*
  按立即值右移。
- 第5-48 页的*VQ{R}SHR{U}N(按立即数)*
  按立即值右移并进行饱和。
- 第5-49 页的*VSLI 和VSRI*
  左移并插入,右移并插入。

5.7.1 VSHL、VQSHL、VQSHLU 和 VSHLL（按立即数）

向量左移、向量饱和左移和向量长型左移。

这些指令获取整数向量中的每个元素，按立即值对其进行左移，并将结果存放到目标向量中。

对于 `VSHL`，每个元素中从左侧移出的位将丢失。

对于 `VQSHL` 和 `VQSHLU`，在进行饱和的情况下将设置粘性 QC 标记（FPSCR 位 [27]）。

对于 `VSHLL`，将用符号或零对值进行扩展。

语法

V{Q}SHL{U}{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{Q}SHL{U}{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

VSHLL{*cond*}.*datatype Qd*, *Dm*, *#imm*

其中：

Q
: 如果存在，则指示在任何结果溢出时对其进行饱和。

U
: 仅当 Q 存在时才适用。指示结果无符号，即使操作数有符号也是如此。

cond
: 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype
: 必须是下列值之一：

I8、I16、I32、I64	对于 VSHL
S8、S16、S32	对于 VSHLL、VQSHL 或 VQSHLU
U8、U16、U32	对于 VSHLL 或 VQSHL
S64	对于 VQSHL 或 VQSHLU
U64	对于 VQSHL。

Qd、*Qm*
: 是四字运算的目标向量和操作数向量。

Dd、*Dm*
: 是双字运算的目标向量和操作数向量。

Qd、*Dm*
: 是长型运算的目标向量和操作数向量。

imm	是指定移位大小的立即数，位于以下范围内： • 对于 VSHLL1 到 size(*datatype*)，为 • 对于 VSHL、VQSHL 或 VQSHLU，为 1 到 (size(*datatype*) - 1)。 允许为 0，但结果代码将反汇编为 VMOV 或 VMOVL。

5.7.2 V{Q}{R}SHL（按有符号变量）

VSHL（向量按有符号变量左移）获取一个向量中的每个元素，按另一个向量的相应元素的最低有效字节中的值对其进行移位，并将结果存放到目标向量中。如果移位值为正数，则该运算为左移。否则为右移。

可以选择对结果执行饱和或舍入运算，或者同时执行这两种运算。如果进行饱和，则会设置粘性 QC 标记（FPSCR 位 [27]）。

语法

```
V{Q}{R}SHL{cond}.datatype {Qd}, Qn, Qm
V{Q}{R}SHL{cond}.datatype {Dd}, Dn, Dm
```

其中：

Q	如果存在，则指示在任何结果溢出时对其进行饱和。
R	如果存在，则指示对每个结果进行舍入。否则将每个结果截断。
cond	是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。
datatype	必须为 S8、S16、S32、S64、U8, U16、U32 或 U64 之一。
Qd、*Qn*、*Qm*	是四字运算的目标向量、第一个操作数向量和第二个操作数向量。
Dd、*Dn*、*Dm*	是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

5.7.3 V{R}SHR{N}、V{R}SRA（按立即数）

V{R}SHR{N}（向量按立即值右移）获取向量中的每个元素，按立即值对其进行右移，并将结果存放到目标向量中。可以选择对结果执行舍入或窄型运算，或者同时执行这两种运算。

V{R}SRA（向量按立即值右移并累加）获取向量中的每个元素，按立即值对其进行右移，并将结果累加到目标向量中。可以选择对结果进行舍入。

语法

V{R}SHR{*cond*}.*datatype* {*Qd*}, *Qm*, #*imm*

V{R}SHR{*cond*}.*datatype* {*Dd*}, *Dm*, #*imm*

V{R}SRA{*cond*}.*datatype* {*Qd*}, *Qm*, #*imm*

V{R}SRA{*cond*}.*datatype* {*Dd*}, *Dm*, #*imm*

V{R}SHRN{*cond*}.*datatype* *Dd*, *Qm*, #*imm*

其中：

R　如果存在，则指示对结果进行舍入。否则将结果截断。

cond　是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype　必须是下列值之一：

S8、S16、S32、S64	对于 V{R}SHR 或 V{R}SRA
U8、U16、U32、U64	对于 V{R}SHR 或 V{R}SRA
I16、I32、I64	对于 V{R}SHRN。

Qd、*Qm*　是四字运算的目标向量和操作数向量。

Dd、*Dm*　是双字运算的目标向量和操作数向量。

Dd、*Qm*　是窄型运算的目标向量和操作数向量。

imm　是指定移位大小的立即数，位于范围 0 到 (size(*datatype*) - 1) 内。

5.7.4 VQ{R}SHR{U}N（按立即数）

VQ{R}SHR{U}N（向量饱和右移、窄型、按立即值，可选舍入）获取整数四字向量中的每个元素，按立即值对其进行右移，并将结果存放到双字向量中。

如果进行饱和，则会设置粘性 QC 标记（FPSCR 位 [27]）。

语法

VQ{R}SHR{U}N{*cond*}.*datatype Dd*, *Qm*, #*imm*

其中：

R
: 如果存在，则指示对结果进行舍入。否则将结果截断。

U
: 如果存在，则指示结果无符号，即使操作数有符号也是如此。否则，结果将与操作数类型相同。

cond
: 是一个可选的条件代码（请参阅第 5-9 页的*条件代码*）。

datatype
: 必须是下列值之一：

S16、S32、S64	对于 VQ{R}SHRN 或 VQ{R}SHRUN
U16、U32、U64	仅限 VQ{R}SHRN。

Dd、*Qm*
: 是目标向量和操作数向量。

imm
: 是指定移位大小的立即数，位于范围 0 到 (size(*datatype*) - 1) 内。

5.7.5 VSLI 和 VSRI

VSLI（向量左移并插入）获取向量中的每个元素，按立即值对其进行左移，并将结果插入目标向量中。每个元素中从左侧移出的位将丢失。

VSRI（向量右移并插入）获取向量中的每个元素，按立即值对其进行右移，并将结果插入目标向量中。每个元素中从最右侧移出的位将丢失。

语法

V*op*{*cond*}.*size* {*Qd*}, *Qm*, #*imm*

V*op*{*cond*}.*size* {*Dd*}, *Dm*, #*imm*

其中:

op 必须为 SLI 或 SRI。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

size 必须为 8、16、32 或 64 之一。

Qd、*Qm* 是四字运算的目标向量和操作数向量。

Dd、*Dm* 是双字运算的目标向量和操作数向量。

imm 是指定移位大小的立即数，位于以下范围内:

- 对于 VSLI，为 0 到 (*size* - 1)
- 对于 VSRI，为 1 到 *size*。

5.8 NEON 通用算术指令

本节包括以下小节:

- 第5-51 页的*VABA{L} 和 VABD{L}*
  向量差值绝对值累加和差值绝对值。
- 第5-52 页的*V{Q}ABS 和 V{Q}NEG*
  向量绝对值和求反。
- 第5-53 页的*V{Q}ADD*、*VADDL*、*VADDW*、*V{Q}SUB*、*VSUBL 和 VSUBW*
  向量加法和减法。
- 第5-54 页的*V{R}ADDHN 和 V{R}SUBHN*
  选择高半部分的向量加法和选择高半部分的向量减法。
- 第5-55 页的*V{R}HADD 和 VHSUB*
  向量半加和半减。
- 第5-56 页的*VPADD{L}*、*VPADAL*
  向量按对加，向量按对加并累加。
- 第5-58 页的*VMAX*、*VMIN*、*VPMAX 和 VPMIN*
  向量最大值，向量最小值，向量按对最大值和向量按对最小值。
- 第5-59 页的*VCLS*、*VCLZ 和 VCNT*
  向量前导符号位计数，前导零计数和设置位计数。
- 第5-60 页的*VRECPE 和 VRSQRTE*
  向量近似倒数和近似平方根倒数。
- 第5-61 页的*VRECPS 和 VRSQRTS*
  向量倒数步进和平方根倒数步进。

5.8.1 VABA{L} 和 VABD{L}

VABA（向量差值绝对值累加）用一个向量的元素减去另一个向量的相应元素，并将结果的绝对值累加到目标向量的元素中。

VABD（向量差值绝对值）用一个向量的元素减去另一个向量的相应元素，并将结果的绝对值存放到目标向量的元素中。

这两个指令的长型格式都可用。

语法

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype* *Qd*, *Dn*, *Dm*

其中：

op 必须为 ABA 或 ABD。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须是下列值之一：

- 对于 VABA、VABAL 或 VABDL，为 S8、S16、S32、U8、U16 或 U32
- 对于 VABD，为 S8、S16、S32、U8、U16、U32 或 F32。

Qd、*Qn*、*Qm* 是四字运算的目标向量、第一个操作数向量和第二个操作数向量。

Dd、*Dn*、*Dm* 是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

Qd、*Dn*、*Dm* 是长型运算的目标向量、第一个操作数向量和第二个操作数向量。

5.8.2 V{Q}ABS 和 V{Q}NEG

VABS（向量绝对值）获取一个向量中每个元素的绝对值，并将结果存放到另一个向量中。（对于浮点格式，仅清除符号位。）

VNEG（向量求反）对一个向量中的每个元素执行求反运算，并将结果存放到另一个向量中。（对于浮点格式，仅反转符号位。）

这两个指令的饱和格式都可用。如果进行饱和，则会设置粘性 QC 标记（FPSCR 位 [27]）。

语法

V{Q}*op*{*cond*}.*datatype Qd*, *Qm*

V{Q}*op*{*cond*}.*datatype Dd*, *Dm*

其中：

Q　　如果存在，则指示在任何结果溢出时对其进行饱和。

op　　必须为 ABS 或 NEG。

cond　　是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype　　必须是下列值之一：

S8、S16、S32	对于 VABS、VNEG、VQABS 或 VQNEG
F32	仅限 VABS 和 VNEG。

Qd、*Qm*　　是四字运算的目标向量和操作数向量。

Dd、*Dm*　　是双字运算的目标向量和操作数向量。

5.8.3 V{Q}ADD、VADDL、VADDW、V{Q}SUB、VSUBL 和 VSUBW

VADD（向量加法）将两个向量中的相应元素相加，并将结果存放到目标向量中。

VSUB（向量减法）用一个向量的元素减去另一个向量的相应元素，并将结果存放到目标向量的结果中。

饱和、长型和宽型格式都可用。如果进行饱和，则会设置粘性 QC 标记（FPSCR 位 [27]）。

语法

V{Q}*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{Q}*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype* *Qd*, *Dn*, *Dm*

V*op*W{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*

其中:

Q　如果存在，则指示在任何结果溢出时对其进行饱和。

op　必须为 ADD 或 SUB。

cond　是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype　必须是下列值之一:

I8、I16、I32、I64、F32 对于 VADD 或 VSUB

S8、S16、S32　对于 VQADD、VQSUB、VADDL、VADDW、VSUBL 或 VSUBW

U8、U16、U32　对于 VQADD、VQSUB、VADDL、VADDW、VSUBL 或 VSUBW

S64、U64　对于 VQADD 或 VQSUB。

Qd、*Qn*、*Qm*　是四字运算的目标向量、第一个操作数向量和第二个操作数向量。

Dd、*Dn*、*Dm*　是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

Qd、*Dn*、*Dm*　是长型运算的目标向量、第一个操作数向量和第二个操作数向量。

Qd、*Qn*、*Dm*　是宽型运算的目标向量、第一个操作数向量和第二个操作数向量。

5.8.4 V{R}ADDHN 和 V{R}SUBHN

V{R}ADDH（向量窄型加法，选择高半部分）将两个向量中的相应元素相加，选择相加结果的最高有效半部，并将最终结果存放到目标向量中。可将结果舍入或截断。

V{R}SUBH（向量窄型减法，选择高半部分）用一个向量的元素减去另一个向量的相应元素，选择相减结果的最高有效半部，并将最终结果存放到目标向量中。可将结果舍入或截断。

语法

V{R}*op*HN{*cond*}.*datatype Dd*, *Qn*, *Qm*

其中:

R　如果存在，则指示对每个结果进行舍入。否则将每个结果截断。

op　必须为 ADD 或 SUB。

cond　是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype　必须为 I16、I32 或 I64 之一。

Dd、*Qn*、*Qm*　是目标向量、第二个操作数向量和第二个操作数向量。

5.8.5 V{R}HADD 和 VHSUB

VHADD（向量半加）将两个向量中的相应元素相加，将每个结果右移一位，并将这些结果存放到目标向量中。可将结果舍入或截断。

VHSUB（向量半减）用一个向量的元素减去另一个向量的相应元素，将每个结果右移一位，并将这些结果存放到目标向量中。结果将始终被截断。

语法

V{R}HADD{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{R}HADD{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VHSUB{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VHSUB{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

其中:

R 如果存在，则指示对每个结果进行舍入。否则将每个结果截断。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 S8、S16、S32、U8、U16 或 U32 之一。

Qd、*Qn*、*Qm* 是四字运算的目标向量、第一个操作数向量和第二个操作数向量。

Dd、*Dn*、*Dm* 是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

5.8.6 VPADD{L}、VPADAL

VPADD（向量按对加）将两个向量的相邻元素对相加，并将结果存放到目标向量中。

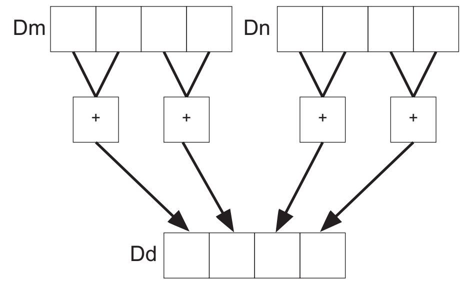


图5-5 VPADD 运算的示例（在本例中，数据类型为 I16）

VPADDL（向量长型按对加）将向量中相邻的元素对相加，用符号或零将结果扩展为原宽度的两倍，并将最终结果存放到目标向量中。

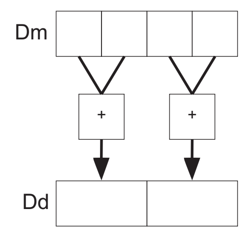


图5-6 双字 VPADDL 运算的示例（在本例中，数据类型为 S16）

VPADAL（向量长型按对加累加）将向量中相邻的元素对相加，并将结果的绝对值累加到目标向量的元素中。

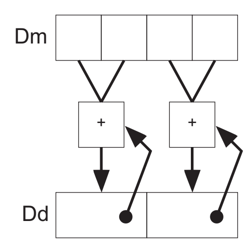


图5-7 VPADAL 运算的示例（在本例中，数据类型为 S16）

语法

```
VPADD{cond}.datatype {Dd}, Dn, Dm
VPopL{cond}.datatype Qd, Qm
VPopL{cond}.datatype Dd, Dm
```

其中:

op 必须为 ADD 或 ADA。

cond 是一个可选的条件代码(请参阅第5-9 页的*条件代码*)。

datatype 必须是下列值之一:

I8、I16、I32和F32	对于 VPADD
S8、S16和S32	对于 VPADDL 或 VPADAL
U8、U16和U32	对于 VPADDL 或 VPADAL。

Dd、*Dn*、*Dm* 是 VPADD 运算的目标向量、第一个操作数向量和第二个操作数向量。

Qd、*Qm* 是四字 VPADDL 或 VPADAL 的目标向量和操作数向量。

Dd、*Dm* 是双字 VPADDL 或 VPADAL 的目标向量和操作数向量。

5.8.7 VMAX、VMIN、VPMAX 和 VPMIN

VMAX（向量最大值）对两个向量中的相应元素进行比较，并将每一对中的较大值复制到目标向量的相应元素中。

VMIN（向量最小值）对两个向量中的相应元素进行比较，并将每一对中的较小值复制到目标向量的相应元素中。

VPMAX（向量按对最大值）对两个向量中的相邻元素对进行比较，并将每一对中的较大值复制到目标向量的相应元素中。操作数和结果必须为双字向量。

VPMIN（向量按对最小值）对两个向量中的相邻元素对进行比较，并将每一对中的较小值复制到目标向量的相应元素中。操作数和结果必须为双字向量。

有关按对运算的图示，请参阅第5-56 页的图5-5。

语法

V*op*{*cond*}.*datatype Qd*, *Qn*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dn*, *Dm*

VP*op*{*cond*}.*datatype Dd*, *Dn*, *Dm*

其中:

op 必须为 MAX 或 MIN。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 S8、S16、S32、U8、U16、U32 或 F32 之一。

Qd、*Qn*、*Qm* 是四字运算的目标向量、第一个操作数向量和第二个操作数向量。

Dd、*Dn*、*Dm* 是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

浮点最大值和最小值

max(+0.0, -0.0) = +0.0.

min(+0.0, -0.0) = -0.0

如果任意输入为非数字，则对应的结果元素为缺省非数字。

5.8.8 VCLS、VCLZ 和 VCNT

VCLS（向量前导符号位计数）计算一个向量的每个元素中最高位后面与最高位相同的连续位数目，并将结果存放到另一个向量中。

VCLZ（向量前导零计数）计算一个向量的每个元素中从最高位开始算起的连续零数目，并将结果存放到另一个向量中。

VCNT（向量设置位计数）计算一个向量的每个元素中位为 1 的数目，并将结果存放到另一个向量中。

语法

V*op*{*cond*}.*datatype Qd*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dm*

其中：

op 必须为 CLS、CLZ 或 CNT 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须是下列值之一：
- 对于 CLS，为 S8、S16 或 S32。
- 对于 CLZ，为 I8、I16 或 I32。
- 对于 CNT，为 I8。

Qd、*Qm* 是四字运算的目标向量和操作数向量。

Dd、*Dm* 是双字运算的目标向量和操作数向量。

5.8.9 VRECPE 和 VRSQRTE

VRECPE（向量近似倒数）求出一个向量中每个元素的近似倒数，并将结果存放到另一个向量中。

VRSQRTE（向量近似平方根倒数）求出一个向量中每个元素的近似平方根倒数，并将结果存放到另一个向量中。

语法

`V`*`op`*`{`*`cond`*`}.`*`datatype Qd`*`, `*`Qm`*

`V`*`op`*`{`*`cond`*`}.`*`datatype Dd`*`, `*`Dm`*

其中：

op 必须为 RECPE 或 RSQRTE。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 U32 或 F32。

Qd、*Qm* 是四字运算的目标向量和操作数向量。

Dd、*Dm* 是双字运算的目标向量和操作数向量。

超出范围的输入的结果

表5-9 显示了超出范围的输入值的结果。

表5-9 超出范围的输入的结果

	操作数元素 (VRECPE)	操作数元素 (VRSQRTE)	结果元素
整数	<= 0x7FFFFFFF	<= 0x3FFFFFFF	0xFFFFFFFF
浮点数	非数字	非数字、负标准数、负无穷大	缺省非数字
	负 0、负非正规数	负 0、负非正规数	负无穷大[a]
	正 0、正非正规数	正 0、正非正规数	正无穷大[a]
	正无穷大	正无穷大	正 0
	负无穷大		负 0

a. 设置 FPSCR (FPSCR[1]) 中的除零异常位

5.8.10 VRECPS 和 VRSQRTS

VRECPS（向量倒数步进）将一个向量的元素与另一个向量的相应元素相乘，用 2 减去每个相乘结果，并将最终结果存放到目标向量的元素中。

VRSQRTS（向量平方根倒数步进）将一个向量的元素与另一个向量的相应元素相乘，用 3 减去每个相乘结果，再将这些结果除以 2，并将最终结果存放到目标向量的元素中。

语法

V*op*{*cond*}.F32 {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.F32 {*Dd*}, *Dn*, *Dm*

其中:

op 必须为 RECPS 或 RSQRTS。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Qd、*Qn*、*Qm* 是四字运算的目标向量、第一个操作数向量和第二个操作数向量。

Dd、*Dn*、*Dm* 是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

超出范围的输入的结果

表5-10 显示了超出范围的输入值的结果。

表5-10 超出范围的输入的结果

第一个操作数元素	第二个操作数元素	结果元素 (VRECPS)	结果元素 (VRSQRTS)
非数字	-	缺省非数字	缺省非数字
-	非数字	缺省非数字	缺省非数字
+/- 0.0 或非正规数	+/- 无穷大	2.0	1.5
+/- 无穷大	+/- 0.0 或非正规数	2.0	1.5

用法

牛顿-拉弗森迭代法：

$x_{n+1} = x_n(2-dx_n)$

如果 x_0 是将 VRECPE 应用于 d 的结果，则收敛到 $(1/d)$。

牛顿-拉弗森迭代法：

$x_{n+1} = x_n(3-dx_n^2)/2$

如果 x_0 是将 VRSQRTE 应用于 d 的结果，则收敛到 $(1/\sqrt{d})$。

5.9 NEON 乘法指令

本节包括以下小节:

- 第5-64 页的*VMUL{L}*、*VMLA{L} 和 VMLS{L}*。

  向量乘法、向量乘加和向量乘减。

- 第5-65 页的*VMUL{L}*、*VMLA{L} 和 VMLS{L}(按标量)*。

  向量乘法、向量乘加和向量乘减(按标量)。

- 第5-66 页的*VQDMULL*、*VQDMLAL 和 VQDMLSL(按向量或标量)*

  向量饱和加倍乘法、向量乘加和向量乘减(按向量或标量)。

- 第5-67 页的*VQ{R}DMULH(按向量或标量)*

  返回高半部分的向量饱和加倍乘法(按向量或标量)。

5.9.1 VMUL{L}、VMLA{L} 和 VMLS{L}

VMUL（向量乘法）将两个向量中的相应元素相乘，并将结果存放到目标向量中。

VMLA（向量乘加）将两个向量中的相应元素相乘，并将结果累加到目标向量的元素中。

VMLS（向量乘减）将两个向量中的相应元素相乘，从目标向量的相应元素中减去相乘的结果，并将最终结果放入目标向量中。

语法

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype* *Qd*, *Dn*, *Dm*

其中:

op 必须是下列值之一：

MUL	乘法
MLA	乘加
MLS	乘减。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须是下列值之一：

I8、I16、I32、F32	对于 MUL、MLA 或 MLS
S8, S16, S32	对于 MULL、MLAL 或 MLSL
U8, U16, U32	对于 MULL、MLAL 或 MLSL
P8	对于 MUL 或 MULL。

有关数据类型 P8 的信息，请参阅第5-16 页的*{0,1} 上的多项式算法*。

Qd、*Qn*、*Qm* 是四字运算的目标向量、第一个操作数向量和第二个操作数向量。

Dd、*Dn*、*Dm* 是双字运算的目标向量、第一个操作数向量和第二个操作数向量。

Qd、*Dn*、*Dm* 是长型运算的目标向量、第一个操作数向量和第二个操作数向量。

5.9.2 VMUL{L}、VMLA{L} 和 VMLS{L}（按标量）

VMUL（向量乘以标量）将向量中的每个元素乘以标量，并将结果放入目标向量中。

VMLA（向量乘加）将两个向量中的相应元素相乘，并将结果累加到目标向量的元素中。

VMLS（向量乘减）将两个向量中的相应元素相乘，从目标向量的相应元素中减去相乘的结果，并将最终结果放入目标向量中。

语法

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*[*x*]

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*[*x*]

V*op*L{*cond*}.*datatype* *Qd*, *Dn*, *Dm*[*x*]

其中：

op 必须是下列值之一：

MUL	乘法
MLA	乘加
MLS	乘减。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须是下列值之一：

I16、I32、F32	对于 MUL、MLA 或 MLS
S16、S32	对于 MULL、MLAL 或 MLSL
U16，U32	对于 MULL、MLAL 或 MLSL。

Qd、*Qn* 是四字运算的目标向量和第一个操作数向量。

Dd、*Dn* 是双字运算的目标向量和第一个操作数向量。

Qd、*Dn* 是长型运算的目标向量和第一个操作数向量。

Dm[*x*] 是存放第二个操作数的标量。

5.9.3 VQDMULL、VQDMLAL 和 VQDMLSL（按向量或标量）

向量饱和加倍乘法指令将其操作数相乘并将结果加倍。VQDMULL 将结果存放到目标寄存器中。VQDMLAL 将结果与目标寄存器中的值相加。VQDMLSL 用目标寄存器中的值减去结果。

如果任意结果溢出，则会对其进行饱和。如果进行饱和，则会设置粘性 QC 标记（FPSCR 位 [27]）。

语法

VQD*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

VQD*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*[*x*]

其中：

op 必须是下列值之一：

MUL	乘法
MLA	乘加
MLS	乘减。

cond 是一个可选的条件代码（请参阅第 5-9 页的*条件代码*）。

datatype 必须为 S16 或 S32。

Qd、*Dn* 是目标向量和第一个操作数向量。

Dm 对于*按向量*运算，是存放第二个操作数的向量。

Dm[*x*] 对于*按标量*运算，是存放第二个操作数的标量。

5.9.4 VQ{R}DMULH（按向量或标量）

向量饱和加倍乘法指令将其操作数相乘并将结果加倍。此类指令仅返回结果的高半部分。

如果任意结果溢出，则会对其进行饱和。如果进行饱和，则会设置粘性 QC 标记（FPSCR 位 [27]）。

语法

```
VQ{R}DMULH{cond}.datatype {Qd}, Qn, Qm
VQ{R}DMULH{cond}.datatype {Dd}, Dn, Dm
VQ{R}DMULH{cond}.datatype {Qd}, Qn, Dm[x]
VQ{R}DMULH{cond}.datatype {Dd}, Dn, Dm[x]
```

其中:

R　如果存在，则指示对每个结果进行舍入。否则将每个结果截断。

cond　是一个可选的条件代码（请参阅第 5-9 页的*条件代码*）。

datatype　必须为 S16 或 S32。

Qd、*Qn*　是四字运算的目标向量和第一个操作数向量。

Dd、*Dn*　是双字运算的目标向量和第一个操作数向量。

Qm 或 *Dm*　对于*按向量*运算，是存放第二个操作数的向量。

Dm[*x*]　对于*按标量*运算，是存放第二个操作数的标量。

5.10 NEON 加载/存储元素和结构指令

本节包括以下小节:

- *交叉存取*。
- 第5-69 页的*加载/存储元素和结构指令中的对齐限制*。
- 第5-70 页的*VLDn 和 VSTn(单个n 元素结构到一条向量线)*。

  此类指令几乎可用于所有数据访问。可加载标准向量 ($n = 1$)。
- 第5-72 页的*VLDn(单个n 元素结构到所有向量线)*。
- 第5-74 页的*VLDn 和 VSTn(多个n 元素结构)*。

5.10.1 交叉存取

此组中的许多指令在将结构存储到内存时提供交叉存取功能，并在从内存加载结构时提供反向交叉存取功能。图5-8 显示了一个反向交叉存取示例。交叉存取就是反向的过程。

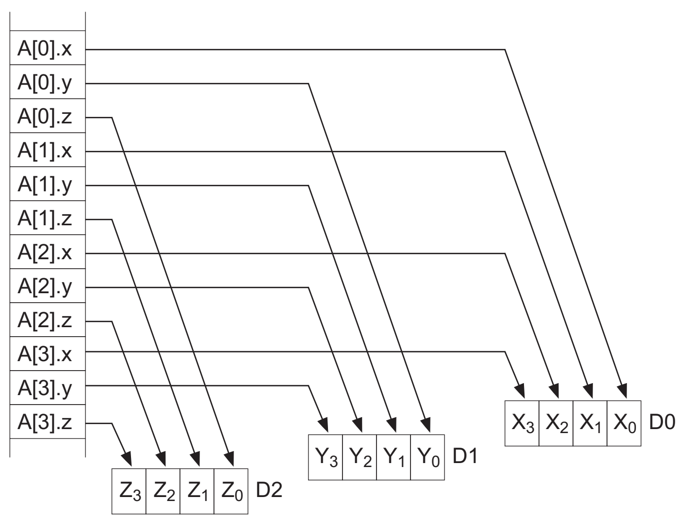


图5-8 对 3 元素数组结构执行反向交叉存取

5.10.2 加载/存储元素和结构指令中的对齐限制

其中许多指令允许指定内存对齐限制。如果指令中未指定对齐，则由 A 位（CP15 寄存器 1 位 [1]）控制对齐限制。

- 如果 A 位为 0，则没有对齐限制（但强序或设备内存除外，其访问必须对齐元素，否则会出现意外结果）。
- 如果 A 位为 1，则访问必须对齐元素。

如果地址未正确对齐，则会发生对齐故障。

5.10.3 VLD*n* 和 VST*n*（单个 *n* 元素结构到一条向量线）

向量加载（单个 *n* 元素结构到一个向量线）将一个 *n* 元素结构从内存加载到一个或多个 NEON 寄存器。未加载的寄存器元素将保持不变。

语法

```
Vopn{cond}.datatype list, [Rn{@align}]{!}

Vopn{cond}.datatype list, [Rn{@align}], Rm
```

其中:

op 必须为 LD 或 ST。

n 必须为 1、2、3 或 4 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 请参阅第5-71 页的表5-11。

list 指定 NEON 寄存器列表。有关选项，请参阅第5-71 页的表5-11。

Rn 是包含基址的 ARM 寄存器。*Rn* 不得为 R15。

align 指定可选对齐。有关选项，请参阅第5-71 页的表5-11。

! 如果 ! 存在，则将 *Rn* 更新为（*Rn* + 指令传送的字节数）。在完成所有加载/存储后，执行该更新。

Rm 是一个包含基址偏移量的 ARM 寄存器。如果 *Rm* 存在，则在使用该地址访问内存*之后*，将 *Rn* 更新为 (*Rn* + *Rm*)。*Rm* 不得为 R13 或 R15。

表5-11 允许的参数组合

n	*datatype*	list [a]	*align* [b]	alignment
1	8	`{Dd[x]}`	-	仅限标准对齐规则
	16	`{Dd[x]}`	`@16`	2 字节
	32	`{Dd[x]}`	`@32`	4 字节
2	8	`{Dd[x], D(d+1)[x]}`	`@16`	2 字节
	16	`{Dd[x], D(d+1)[x]}`	`@32`	4 字节
		`{Dd[x], D(d+2)[x]}`	`@32`	4 字节
	32	`{Dd[x], D(d+1)[x]}`	`@64`	8 字节
		`{Dd[x], D(d+2)[x]}`	`@64`	8 字节
3	8、16 或 32	`{Dd[x], D(d+1)[x], D(d+2)[x]}`	-	仅限标准对齐规则
		`{Dd[x], D(d+2)[x], D(d+4)[x]}`	-	仅限标准对齐规则
4	8	`{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}`	`@32`	4 字节
		`{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}`	`@32`	4 字节
	16	`{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}`	`@64`	8 字节
		`{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}`	`@64`	8 字节
	32	`{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}`	`@64` 或 `@128`	8 字节或 16 字节
		`{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}`	`@64` 或 `@128`	8 字节或 16 字节

a. list 中的每个寄存器都必须位于范围 D0 到 D31 内。

b. 可省略 `Align`。在这种情况下，将应用标准对齐规则，详情请参阅第5-69 页的*加载/存储元素和结构指令中的对齐限制*。

5.10.4 VLD*n*（单个 *n* 元素结构到所有向量线）

向量加载（单个 *n* 元素结构到所有向量线）将一个 *n* 元素结构的多个副本从内存加载到一个或多个 NEON 寄存器中。

语法

```
VLDn{cond}.datatype list, [Rn{@align}]{!}
VLDn{cond}.datatype list, [Rn{@align}], Rm
```

其中：

n 必须为 1、2、3 或 4 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 请参阅第5-73 页的表5-12。

list 指定 NEON 寄存器列表。有关选项，请参阅第5-73 页的表5-12。

Rn 是包含基址的 ARM 寄存器。*Rn* 不得为 R15。

align 指定可选对齐。有关选项，请参阅第5-73 页的表5-12。

! 如果 ! 存在，则将 *Rn* 更新为（*Rn* + 指令传送的字节数）。在完成所有加载/存储后，执行该更新。

Rm 是一个包含基址偏移量的 ARM 寄存器。如果 *Rm* 存在，则在使用该地址访问内存*之后*，将 *Rn* 更新为 (*Rn* + *Rm*)。*Rm* 不得为 R13 或 R15。

表5-12 允许的参数组合

n	*datatype*	list [a]	*align* [b]	alignment
1	8	`{Dd[]}`	-	仅限标准对齐规则
		`{Dd[],D(d+1)[]}`	-	仅限标准对齐规则
	16	`{Dd[]}`	`@16`	2 字节
		`{Dd[],D(d+1)[]}`	`@16`	2 字节
	32	`{Dd[]}`	`@32`	4 字节
		`{Dd[],D(d+1)[]}`	`@32`	4 字节
2	8	`{Dd[], D(d+1)[]}`	`@8`	字节
		`{Dd[], D(d+2)[]}`	`@8`	字节
	16	`{Dd[], D(d+1)[]}`	`@16`	2 字节
		`{Dd[], D(d+2)[]}`	`@16`	2 字节
	32	`{Dd[], D(d+1)[]}`	`@32`	4 字节
		`{Dd[], D(d+2)[]}`	`@32`	4 字节
3	8、16 或 32	`{Dd[], D(d+1)[], D(d+2)[]}`	-	仅限标准对齐规则
		`{Dd[], D(d+2)[], D(d+4)[]}`	-	仅限标准对齐规则
4	8	`{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}`	`@32`	4 字节
		`{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}`	`@32`	4 字节
	16	`{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}`	`@64`	8 字节
		`{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}`	`@64`	8 字节
	32	`{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}`	`@64` 或 `@128`	8 字节或 16 字节
		`{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}`	`@64` 或 `@128`	8 字节或 16 字节

a. list 中的每个寄存器都必须位于范围 D0 到 D31 内。

b. 可省略 *`Align`*。在这种情况下，将应用标准对齐规则，详情请参阅第5-69 页的*加载/存储元素和结构指令中的对齐限制*。

5.10.5 VLD*n* 和 VST*n*（多个 *n* 元素结构）

向量加载（多个 *n* 元素结构）使用反向交叉存取功能，将多个 *n* 元素结构从内存加载到一个或多个 NEON 寄存器中（除非 *n* == 1）。会加载每个寄存器的每个元素。

向量存储（多个 *n* 元素结构）使用交叉存取功能将多个 *n* 元素结构从一个或多个 NEON 寄存器存储到内存中（除非 *n* == 1）。会存储每个寄存器的每个元素。

语法

```
Vopn{cond}.datatype list, [Rn{@align}]{!}
```

```
Vopn{cond}.datatype list, [Rn{@align}], Rm
```

其中:

op 必须为 LD 或 ST。

n 必须为 1、2、3 或 4 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 有关选项，请参阅第5-75 页的表5-13。

list 指定 NEON 寄存器列表。有关选项，请参阅第5-75 页的表5-13。

Rn 是包含基址的 ARM 寄存器。*Rn* 不得为 R15。

align 指定可选对齐。有关选项，请参阅第5-75 页的表5-13。

! 如果 ! 存在，则将 *Rn* 更新为（*Rn* + 指令传送的字节数）。在完成所有加载/存储后，执行该更新。

Rm 是一个包含基址偏移量的 ARM 寄存器。如果 *Rm* 存在，则在使用该地址访问内存*之后*，将 *Rn* 更新为 (*Rn* + *Rm*)。*Rm* 不得为 R13 或 R15。

表5-13 允许的参数组合

n	*datatype*	list [a]	*align* [b]	*alignment*
1	8、16、32 或 64	{Dd}	@64	8 字节
		{Dd, D(d+1)}	@64 或 @128	8 字节或 16 字节
		{Dd, D(d+1), D(d+2)}	@64	8 字节
		{Dd, D(d+1), D(d+2), D(d+3)}	@64、@128 或 @256	8 字节、16 字节或 32 字节
2	8、16 或 32	{Dd, D(d+1)}	@64, @128	8 字节或 16 字节
		{Dd, D(d+2)}	@64, @128	8 字节或 16 字节
		{Dd, D(d+1), D(d+2), D(d+3)}	@64、@128 或 @256	8 字节、16 字节或 32 字节
3	8、16 或 32	{Dd, D(d+1), D(d+2)}	@64	8 字节
		{Dd, D(d+2), D(d+4)}	@64	8 字节
4	8、16 或 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64、@128 或 @256	8 字节、16 字节或 32 字节
		{Dd, D(d+2), D(d+4), D(d+6)}	@64、@128 或 @256	8 字节、16 字节或 32 字节

a. list 中的每个寄存器都必须位于范围 D0 到 D31 内。
b. 可省略 `Align`。在这种情况下，将应用标准对齐规则，详情请参阅第5-69 页的*加载/存储元素和结构指令中的对齐限制*。

5.11 NEON 和 VFP 伪指令

本节包括以下小节:

5.11.1 VLDR 伪指令

`VLDR` 伪指令将一个常数值加载到 64 位 NEON 向量的每个元素，或者加载到 VFP 单精度或双精度寄存器。

——注意——

本节仅介绍 `VLDR` *伪*指令。有关 `VLDR` *指令*的信息，请参阅第5-19 页的*VLDR 和 VSTR*。

语法

`VLDR{`*`cond`*`}.`*`datatype Dd`*`,=`*`constant`*

`VLDR{`*`cond`*`}.`*`datatype Sd`*`,=`*`constant`*

其中：

datatype 必须是下列值之一：

`I`*`n`*	仅限 NEON
`S`*`n`*	仅限 NEON
`U`*`n`*	仅限 NEON
`F32`	NEON 或 VFP
`F64`	仅限 VFP

n 必须为 8、16、32 或 64 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Dd 或 *Sd* 是要加载的扩展寄存器。

constant 是 *datatype* 的相应类型的常数。

用法

如果某一指令（如 `VMOV`）可用于直接将常数生成到寄存器中，则汇编程序将使用该指令。 否则，汇编程序生成一个包含常数的双字文字池条目，并使用 `VLDR` 指令加载该常数。

5.11.2 VMOV2

`VMOV2` 伪指令生成一个常数并将其存放到 NEON 向量的每个元素中，而不从文字池中加载值。它始终正好汇编为两个指令。

`VMOV2` 可生成任何 16 位常数，以及限定范围的 32 位和 64 位常数。

语法

`VMOV2{cond}.datatype Qd, #constant`

`VMOV2{cond}.datatype Dd, #constant`

其中：

datatype 必须是下列值之一：

- `I8`、`I16`、`I32` 或 `I64`
- `S8`、`S16`、`S32` 或 `S64`
- `U8`、`U16`、`U32` 或 `U64`
- `F32`。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Qd 或 *Dd* 是要加载的扩展寄存器。

constant 是 *datatype* 的相应类型的常数。

用法

`VMOV2` 通常汇编为 `VMOV` 或 `VMVN` 指令，后跟 `VBIC` 或 `VORR` 指令。有关详细信息，请参阅第5-37 页的*VMOV、VMVN（立即数）*和第5-27 页的*VBIC 和 VORR（立即数）*。

5.11.3 VAND 和 VORN（立即数）

VAND（立即数按位与）获取目标向量的每个元素，对这些元素和一个立即数执行按位与运算，并将结果返回到目标向量。

VORN（立即数按位或非）获取目标向量的每个元素，对这些元素和一个立即数执行按位或补运算，并将结果返回到目标向量。

——注意——

在反汇编时，这些伪指令将反汇编为相应的 VBIC 和 VORR 指令，并使用立即数补码。

语法

Vop{*cond*}.*datatype Qd*, #*imm*

Vop{*cond*}.*datatype Dd*, #*imm*

其中:

op 必须为 VAND 或 VORN。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 I16 或 I32。

Qd 或 *Dd* 是用于存放结果的 NEON 寄存器。

imm 是立即数。

立即数

如果 *datatype* 为 I16，则立即数必须采用下列格式之一：

- 0xFFXY
- 0xXYFF。

如果 *datatype* 为 I32，则立即数必须采用下列格式之一：

- 0xFFFFFFXY
- 0xFFFFXYFF
- 0xFFXYFFFF
- 0xXYFFFFFF。

5.11.4 VACLE 和 VACLT

向量绝对值比较获取一个向量中每个元素的绝对值，并将其与另一个向量中相应元素的绝对值进行比较。如果条件为 True，则将目标向量中的相应元素全部设置为 1。否则，全部设置为 0。

——注意——

在反汇编时，这些伪指令将反汇编为相应的 `VACGE` 和 `VACGT` 指令，并反转操作数。

语法

VAC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VAC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

其中：

op 必须是下列值之一：

- `LE` 绝对值小于或等于
- `LT` 绝对值小于。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 `F32`。

Qd 或 *Dd* 是用于存放结果的 NEON 寄存器。

结果的数据类型为 `I32`。

Qn 或 *Dn* 是用于存放第一个操作数的 NEON 寄存器。

Qm 或 *Dm* 是用于存放第二个操作数的 NEON 寄存器。

5.11.5 VCLE 和 VCLT

向量比较获取向量中每个元素的值，并将其与另一个向量中相应元素的值或零进行比较。如果条件为 True，则将目标向量中的相应元素全部设置为 1。否则，全部设置为 0。

———注意———

在反汇编时，这些伪指令将反汇编为相应的 VCGE 和 VCGT 指令，并反转操作数。

——————————

语法

VC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

其中:

op 必须是下列值之一:

LE 小于或等于

LT 小于。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

datatype 必须为 S8、S16、S32、U8、U16、U32 或 F32 之一。

Qd 或 *Dd* 是用于存放结果的 NEON 寄存器。

结果的数据类型为:

- 对于操作数数据类型 I32、S32、U32 或 F32，为 I32
- 对于操作数类型 I16、S16 或 U16，为 I16。
- 对于操作数数据类型 I8、S8 或 U8 ,为 I8。

Qn 或 *Dn* 是用于存放第一个操作数的 NEON 寄存器。

Qm 或 *Dm* 是用于存放第二个操作数的 NEON 寄存器。

5.12 NEON 和 VFP 系统寄存器

以下三个 NEON 和 VFP 系统寄存器可在所有 NEON 和 VFP 执行中使用:

- *FPSCR，浮点状态和控制寄存器*
- 第5-84 页的*FPEXC，浮点异常寄存器*
- 第5-84 页的*FPSID，浮点系统标识寄存器*
- 第5-85 页的*修改 NEON 和 VFP 系统寄存器的个别位*。

NEON 或 VFP 的特定执行方式可以拥有附加的寄存器（请参阅您所使用的 VFP 协处理器的技术参考手册）。

5.12.1 FPSCR，浮点状态和控制寄存器

`FPSCR` 包含所有用户级 NEON 和 VFP 状态位及控制位。NEON 仅使用位 [31:27]。这些位的用法如下所示:

位 [31:28] 是 N、Z、C 和 V 标记。这些是 NEON 和 VFP 状态标记。它们只有在被复制到 CPSR 中的状态标记之后，才能用于控制有条件执行（请参阅第5-9 页的*条件代码*）。

位 [27] 是 QC（累积饱和标记）。如果在 NEON 或 VFP 饱和指令中进行饱和，则会设置此标记。

位 [24] 是清零模式控制位:

0 禁用清零模式。

1 启用清零模式。

根据所使用的硬件和软件，清零模式能以范围损失作为代价，提供更高的性能（请参阅第5-86 页的*清零模式*）。

——注意——

无论此位如何，NEON 都始终使用清零模式。

当要求兼容 IEEE 754 时，不得使用清零模式。

位 [23:22] 按如下如示控制舍入模式:

0b00 *向最接近的数舍入* (RN) 模式。

0b01 *向正无穷大舍入* (RP) 模式。

0b10 *向负无穷大舍入* (RM) 模式。

0b11 *向零舍入* (RZ) 模式。

位 [21:20] `STRIDE` 是向量中连续值之间的距离（请参阅第5-98 页的*向量*）。跨距的控制方式如下：

0b00 `STRIDE` = 1

0b11 `STRIDE` = 2。

位 [18:16] `LEN` 是每个向量所使用的寄存器的数目（请参阅第5-98 页的*向量*）。其值是 1 + 位 [18:16] 的值：

0b000 `LEN` = 1

...

0b111 `LEN` = 8。

位 [12:8] 是异常陷阱启用位：

IXE 启用不精确异常

UFE 启用下溢异常

OFE 启用溢出异常

DZE 启用除零异常

IOE 启用无效运算异常。

本手册不涉及浮点异常捕获的用法。有关信息，请参阅您所使用的 VFP 协处理器的技术参考手册。

位 [4:0] 是累积异常位：

IXC 不精确异常

UFC 下溢异常

OFC 溢出异常

DZC 除零异常

IOC 无效运算异常。

当发生相应的异常时，将设置累积异常位。仅当直接写入 `FPSCR` 来加以清除之后，才会清除累积异常位。

其他所有位 是基本的 NEON 和 VFP 规范中未使用的位。它们可以用在特定执行方式中（请参阅您所使用的 VFP 协处理器的技术参考手册）。除非要遵照特定执行中的任何用法，否则不要修改这些位。

若要更改某些位而不影响其他位，请使用读改写过程（请参阅第5-85 页的*修改 NEON 和 VFP 系统寄存器的个别位*）。

注意

在新代码中已弃用向量模式。`LEN` 和 `STRIDE` 将设置为 1。

5.12.2 FPEXC，浮点异常寄存器

只能在特权模式下访问 FPEXC。该寄存器包含以下各位：

位 [31] 是 EX 位。可以在所有 NEON 或 VFP 执行中读取该位。在有些执行中，还可以写入该位。

如果该位的值为 0，则 NEON 或 VFP 系统中的唯一有效状态是通用寄存器加上 FPSCR 和 FPEXC 的内容。

如果该位的值为 1，则需要与执行相关的信息来保存状态（请参阅您所使用的 VFP 协处理器的技术参考手册）。

位 [30] 是 EN 位。可以在所有 NEON 或 VFP 执行中读写此位。

如果该位的值为 1，则启用 NEON（如果存在）和 VFP（如果存在）协处理器并正常执行运算。

如果该位的值为 0，则禁用 NEON 和 VFP。这两个处理器被禁用后，可以读写 FPSID 或 FPEXC 寄存器，但会将其他 NEON 或 VFP 指令视为未定义的指令。

位 [29:0] 可由 VFP 的特定执行使用。您无需访问这些位就可使用本章介绍的所有 VFP 函数。

除非要遵照特定执行中的用法，否则不要修改这些位（请参阅您所使用的 VFP 协处理器的技术参考手册）。

若要更改某些位而不影响其他位，请使用读改写过程（请参阅第5-85 页的*修改 NEON 和 VFP 系统寄存器的个别位*）。

5.12.3 FPSID，浮点系统标识寄存器

FPSID 是一个只读寄存器。可以读取它来确定您的程序运行在 NEON 或 VFP 体系结构的哪一个执行版本上。

5.12.4 修改 NEON 和 VFP 系统寄存器的个别位

若要更改 NEON 和 VFP 系统寄存器的某些位而不影响其他位，可使用与以下示例类似的读改写过程:

```
VMRS    r10,FPSCR           ; copy FPSCR into r10
BIC     r10,r10,#0x00370000 ; clears STRIDE and LEN
ORR     r10,r10,#0x00030000 ; sets STRIDE = 1, LEN = 4
VMSR    FPSCR,r10           ; copy r10 back into FPSCR
```

请参阅第5-24 页的*VMRS 和 VMSR*。

5.13 清零模式

VFP 的某些执行方式使用支持代码来处理非正规数。在涉及非正规数的计算中，这种系统的性能比其在常规计算中的性能低得多。

清零模式用 0 替代了非正规数。这不符合 IEEE 754 算法，但在某些情况下可大大改进性能。

NEON 和 VFPv3 清零模式保留符号位。VFPv2 清零模式将符号位清为 +0。

NEON 始终使用清零模式。

5.13.1 何时使用清零模式

当符合下列所有条件时，应选择清零模式:

- 对系统没有 IEEE 754 兼容性要求
- 所使用的算法有时会产生非正规数
- 系统使用支持代码来处理非正规数
- 所使用的算法不依赖于其对非正规数的保存精度
- 所使用的算法不会因用 0 替换非正规数而频繁产生异常。

如果不同部分的代码有不同的要求，可以随时在清零模式和正常模式之间转换。寄存器中已存在的数不受模式改变的影响。

5.13.2 使用清零模式的影响

对于某些异常（请参阅第5-87 页的*不受清零模式影响的运算。*），清零模式对浮点运算有下列影响:

- 当非正规数用作浮点运算的输入时，它将被视为 0。源寄存器保持不变。
- 如果在执行舍入之前，单精度浮点运算的结果位于 -2^{-126} 到 $+2^{-126}$ 范围内，则将该结果替换为 0。
- 如果在执行舍入之前，双精度浮点运算的结果位于 -2^{-1022} 到 $+2^{-1022}$ 范围内，则将该结果替换为 0。

只要将非正规数用作操作数或将结果清零，就会发生不精确异常。在清零模式下，不会发生下溢异常。

5.13.3 不受清零模式影响的运算。

即时在清零模式下，也可以对非正规数执行下列 NEON 和 VFP 运算，而不必将结果清零:

- 复制、绝对值和求反（请参阅第5-29 页的*VMOV、VMVN（寄存器）*、第5-89 页的*VABS、VNEG 和 VSQRT* 和第5-52 页的*V{Q}ABS 和 V{Q}NEG*）。
- 复制（请参阅第5-35 页的*VDUP*）。
- 交换（请参阅第5-40 页的*VSWP*）。
- 加载和存储（请参阅第5-19 页的*VLDR 和 VSTR*）。
- 加载多个和存储多个（请参阅第5-20 页的*VLDM、VSTM、VPOP 和 VPUSH*）。
- 在扩展寄存器和 ARM 通用寄存器之间传送（请参阅第5-21 页的*VMOV（在两个ARM 寄存器和一个扩展寄存器之间）*、第5-22 页的*VMOV（在一个 ARM 寄存器和一个NEON 标量之间）*和第5-23 页的*VMOV（在一个ARM 寄存器和一个单精度 VFP 之间）*）。

5.14 VFP 指令

本节包括以下小节:

- 第5-89 页的*VABS、VNEG 和 VSQRT*

  浮点绝对值、求反和平方根。

- 第5-90 页的*VADD、VSUB 和 VDIV*

  浮点加法、减法和除法。

- 第5-91 页的*VMUL、VMLA、VMLS、VNMUL、VNMLA 和 VNMLS*

  浮点乘法和乘加，包含可选求反。

- 第5-92 页的*VCMP*

  浮点数比较。

- 第5-93 页的*VCVT（在单精度数和双精度数之间）*

  在单精度数和双精度数之间转换。

- 第5-94 页的*VCVT（在浮点数和整数之间）*

  在浮点数和整数之间转换。

- 第5-95 页的*VCVT（在浮点数和定点数之间）*

  在浮点数和定点数之间转换。

- 第5-96 页的*VMOV*

  将浮点常数插入单精度或双精度寄存器。

5.14.1 VABS、VNEG 和 VSQRT

浮点绝对值、求反和平方根。

这些指令可以是标量、向量或混合型（请参阅第5-99 页的*VFP 向量和标量运算*）。

语法

`V`*`op`*`{`*`cond`*`}.F32` *`Sd`*`,` *`Sm`*

`V`*`op`*`{`*`cond`*`}.F64` *`Dd`*`,` *`Dm`*

其中:

op 是 `ABS`、`NEG` 或 `SQRT` 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Sd、*Sm* 是用于存放结果和操作数的单精度寄存器。

Dd、*Dm* 是用于存放结果和操作数的双精度寄存器。

用法

`VABS` 指令获取 *Sm* 或 *Dm* 的内容，清除符号位并将结果存放到 *Sd* 或 *Dd* 中。这样就得到绝对值。

`VNEG` 指令获取 *Sm* 或 *Dm* 的内容，更改符号位，并将结果存放到 *Sd* 或 *Dd* 中。这样就得到与原有值符号相反的值。

`VSQRT` 指令获取 *Sm* 或 *Dm* 的内容的平方根，并将结果存放到 *Sd* 或 *Dd* 中。

对于 `VABS` 和 `VNEG` 指令，如果操作数为非数字，则符号位的确定视根据上述各个情况而定，但不会产生异常。

浮点异常

`VABS` 和 `VNEG` 指令不会产生任何异常。

`VSQRT` 指令会产生无效运算或不精确异常。

5.14.2 VADD、VSUB 和 VDIV

浮点加法、减法和除法。

这些指令可以是标量、向量或混合型（请参阅第5-99 页的*VFP 向量和标量运算*）。

语法

```
Vop{cond}.F32 {Sd}, Sn, Sm
Vop{cond}.F64 {Dd}, Dn, Dm
```

其中:

op 是 ADD、SUB 或 DIV 之一。

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Sd、*Sn*、*Sm* 是用于存放结果和操作数的单精度寄存器。

Dd、*Dn*、*Dm* 是用于存放结果和操作数的双精度寄存器。

用法

VADD 指令将操作数寄存器中的值相加，并将结果存放到目标寄存器中。

VSUB 指令用第一个操作数寄存器中的值中减去第二个操作数寄存器中的值，并将结果存放到目标寄存器中。

VDIV 指令用第一个操作数寄存器中的值除以第二个寄存器中的值，并将结果存放到目标寄存器中。

浮点异常

VADD 和 VSUB 指令会产生无效运算、溢出或不精确异常。

FDIV 运算会产生除零、无效运算、溢出、下溢或不精确异常。

5.14.3 VMUL、VMLA、VMLS、VNMUL、VNMLA 和 VNMLS

浮点乘法和乘加，包含可选求反。

这些指令可以是标量、向量或混合型（请参阅第5-99 页的*VFP 向量和标量运算*）。

语法

`V{N}`*`op`*`{`*`cond`*`}.F32` *`Sd`*, *`Sn`*, *`Sm`*

`V{N}`*`op`*`{`*`cond`*`}.F64` *`Dd`*, *`Dn`*, *`Dm`*

其中:

`N` 对最终结果执行求反。

`op` 是 `MUL`、`MLA` 或 `MLS` 之一。

`cond` 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

`Sd`、*`Sn`*、*`Sm`* 是用于存放结果和操作数的单精度寄存器。

`Dd`、*`Dn`*、*`Dm`* 是用于存放结果和操作数的双精度寄存器。

用法

`MUL` 运算将操作数寄存器中的值相乘，并将结果存放到目标寄存器中。

`MLA` 运算将操作数寄存器中的值相乘，将所得的值加到目标寄存器中，并将最终结果放入目标寄存器中。

`MLS` 运算将操作数寄存器中的值相乘，用目标寄存器中的值中减去相乘结果，并将最终结果存放到目标寄存器中。

在上述每一运算中，如果使用了 `N` 选项，则对最终结果执行求反运算。

浮点异常

这些指令会产生无效运算、溢出、下溢、不精确或非标准输入异常。

5.14.4 VCMP

浮点数比较。

VCMP 始终为标量。

语法

```
VCMP{cond}.F32 Sd, Sm
VCMP{cond}.F32 Sd, #0
VCMP{cond}.F64 Dd, Dm
VCMP{cond}.F64 Dd, #0
```

其中:

cond 是一个可选的条件代码(请参阅第5-9 页的*条件代码*)。

Sd、*Sm* 是用于存放操作数的单精度寄存器。

Dd、*Dm* 是用于存放操作数的双精度寄存器。

用法

VCMP 指令用第一个操作数寄存器中的值中减去第二个操作数寄存器中的值(或零,如果第二个操作数为 #0),并在结果上设置 VFP 条件标记(请参阅第5-9 页的*条件代码*)。

浮点异常

VCMP 指令会产生无效运算异常。

5.14.5 VCVT（在单精度数和双精度数之间）

在单精度数和双精度数之间转换。

VCVT 始终为标量。

语法

```
VCVT{cond}.F64.F32 Dd, Sm
```

```
VCVT{cond}.F32.F64 Sd, Dm
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Dd 是用于存放结果的双精度寄存器。

Sm 是用于存放操作数的单精度寄存器。

Sd 是用于存放结果的单精度寄存器。

SD 是用于存放操作数的双精度寄存器。

用法

这些指令将 *Sm* 中的单精度值转换为双精度值，并将结果存放到 *Dd* 中；或将 *Dm* 中的双精度值转换为单精度值，并将结果存放到 *Sd* 中。

浮点异常

这些指令会产生无效运算异常。

5.14.6 VCVT（在浮点数和整数之间）

在浮点数和整数之间转换。

VCVT 始终为标量。

语法

```
VCVT{R}{cond}.type.F64 Sd, Dm
VCVT{R}{cond}.type.F32 Sd, Sm
VCVT{cond}.F64.type Dd, Sm
VCVT{cond}.F32.type Sd, Sm
```

其中:

R	使该运算使用 FPSCR 所指定的舍入模式。否则，该运算将向零舍入。
cond	是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。
type	可以为 U32（无符号 32 位整数）或 S32（有符号 32 位整数）。
Sd	是用于存放结果的单精度寄存器。
Dd	是用于存放结果的双精度寄存器。
Sm	是用于存放操作数的单精度寄存器。
Dm	是用于存放操作数的双精度寄存器。

用法

此指令的前两种形式将浮点数转换为整数。

此指令的后两种形式将整数转换为浮点数。

浮点异常

这些指令会产生非标准输入、无效运算或不精确异常。

5.14.7 VCVT（在浮点数和定点数之间）

在浮点数和定点数之间转换。

VCVT 始终为标量。

语法

VCVT{*cond*}.*type*.F64 *Dd*, *Dd*, #*fbits*

VCVT{*cond*}.*type*.F32 *Sd*, *Sd*, #*fbits*

VCVT{*cond*}.F64.*type* *Dd*, *Dd*, #*fbits*

VCVT{*cond*}.F32.*type* *Sd*, *Sd*, #*fbits*

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

type 可以为以下任一值:

S16	16 位有符号定点数
U16	16 位无符号定点数
S32	32 位有符号定点数
U32	32 位无符号定点数。

Sd 是用于存放操作数和结果的单精度寄存器。

Dd 是用于存放操作数和结果的双精度寄存器。

fbits 是定点数中的小数位数。如果 *type* 为 S16 或 U16，则位于范围 0 到 16 内；如果 *type* 为 S32 或 U32，则位于范围 1 到 32 内。

用法

此指令的前两种形式将浮点数转换为定点数。

此指令的后两种形式将定点数转换为浮点数。

在上述所有运算中，定点数包含在寄存器的最低有效 16 位或 32 位中。

浮点异常

这些指令会产生非标准输入、无效运算或不精确异常。

5.14.8 VMOV

将一个浮点常数插入单精度或双精度寄存器中，或将一个寄存器复制到另一个寄存器中。

此指令始终为标量。

语法

```
VMOV{cond}.F32 Sd, #imm

VMOV{cond}.F64 Dd, #imm

VMOV{cond}.F32 Sd, Sm

VMOV{cond}.F64 Dd, Dm
```

其中:

cond 是一个可选的条件代码（请参阅第5-9 页的*条件代码*）。

Sd 是单精度目标寄存器。

Dd 是双精度目标寄存器。

imm 是浮点常数。

Sm 是单精度源寄存器。

Dm 是双精度源寄存器。

常数值

任何常数均可以 $+/-n * 2^{-}$ 形式表示，其中 *n* 和 *r* 是整数，16 <= *n* <= 31，0 <= r <= 7。

体系结构

此指令可在 VFPv3 中使用。

5.15 VFP 向量模式

大多数算术指令都可应用在这些向量上，从而实现*单指令多数据* (SIMD) 并行处理。此外，浮点加载和存储指令具有多种寄存器形式，从而使向量可在内存之间传送。

有关 VFP 协处理器的更多详细信息，请参阅*《ARM 体系结构参考手册》*。

——注意——

在新代码中已弃用 VFP 向量模式。

5.15.1 寄存器组

VFP 寄存器的组成如下所示：

- 四个由八个单精度寄存器组成的寄存器组：s0 到 s7、s8 到 s15、s16 到 s23 和 s24 到 s31
- 八个（在 VFPv2 中为四个）由四个双精度寄存器组成的寄存器组：d0 到 d3、d4 到 d7、d8 到 d11、d12 到 d15、d16 到 d19、d20 到 d23、d24 到 d27 和 d28 到 d31
- 单精度和双精度寄存器的任意组合。

有关进一步说明，请参阅图5-9 和图5-10。

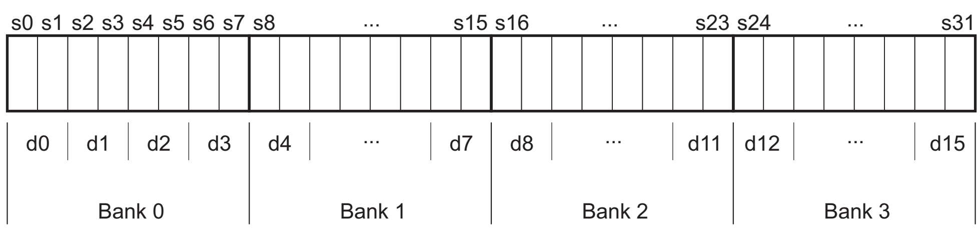


图5-9 VFPv2 寄存器组

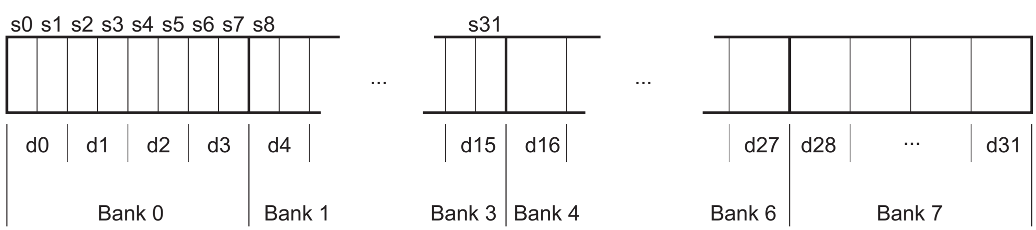


图5-10 VFPv3 寄存器组

5.15.2 向量

一个向量最多可使用同一组中的八个单精度寄存器或四个双精度寄存器。向量所使用的寄存器数目由 FPSCR 中的 LEN 位控制（请参阅第5-82 页的*FPSCR，浮点状态和控制寄存器*）。

向量可以从任何寄存器开始。向量所使用的第一个寄存器在个别指令的寄存器字段中指定。

向量绕回

如果向量扩展至寄存器组的末尾以外，它将绕回到同一组的开始处，例如：

- 从 s5 开始的长度为 6 的向量是 {s5, s6, s7, s0, s1, s2}
- 从 s15 开始的长度为 3 的向量是 {s15, s8, s9}
- 从 s22 开始的长度为 4 的向量是 {s22, s23, s16, s17}
- 从 d7 开始的长度为 2 的向量是 {d7, d4}
- 从 d10 开始的长度为 3 的向量是 {d10, d11, d8}

一个向量不能包含来自多个组的寄存器。

向量跨距

向量可以占用连续的寄存器（如上例所示），也可以占用交替的寄存器。这是由 FPSCR 中的 STRIDE 位来控制的（请参阅第5-82 页的*FPSCR，浮点状态和控制寄存器*）。例如：

- 长度为 3，跨距为 2，从 s1 开始的向量是 {s1, s3, s5}
- 长度为 4，跨距为 2，从 s6 开始的向量是 {s6, s0, s2, s4}
- 长度为 2，跨距为 2，从 d1 开始的向量是 {d1, d3}。

向量长度的限制

向量对同一寄存器不能使用两次。允许向量绕回，这意味着您不能拥有：

- 长度 > 4 且跨距 = 2 的单精度向量
- 长度 > 4 且跨距 = 1 的双精度向量
- 长度 > 2 且跨距 = 2 的双精度向量。

5.15.3 VFP 向量和标量运算

VFP 算术指令可用于：

- 标量
- 向量
- 标量和向量。

使用 FPSCR 中的 LEN 位来控制向量的长度（请参阅第5-82 页的*FPSCR，浮点状态和控制寄存器*）。

当 LEN 为 1 时，所有运算都是标量运算。

向量所具有的*跨距*可以为 1 或 2，这由 FPSCR 中的 STRIDE 位控制。如果 STRIDE 为 1，则向量的元素占用寄存器组中的连续寄存器。如果 STRIDE 为 2，则向量的元素占用寄存器组中的交替寄存器。

控制标量、向量和混合运算

当 LEN 大于 1 时，算术运算的行为取决于目标寄存器和操作数寄存器位于哪个寄存器组中（请参阅第5-97 页的*寄存器组*）。

假设指令采用以下一般形式：

```
Op  Fd,Fn,Fm
Op  Fd,Fm
```

其行为如下：

- 如果 *Fd* 位于第一个或第五个寄存器组（s0 到 s7、d0 到 d3 或 d16 到 d19），则运算为标量运算。
- 如果 *Fm* 位于第一个或第五个寄存器组，但 *Fd* 不在其中，则运算为混合运算。
- 如果 *Fd* 和 *Fm* 都不在第一个或第五个寄存器组，则运算为向量运算。

标量运算

Op 作用于 *Fm* 中的值以及 *Fn* 中的值（如果存在）。结果存放在 *Fd* 中。

向量运算

Op 作用于从 *Fm* 开始的向量中的值以及从 *Fn* 开始的向量中的值（如果存在）。结果存放在从 *Fd* 开始的向量中。

标量和向量混合运算

对于单操作数指令，*Op* 作用于 *Fm* 中的单个值。结果的 LEN 个副本存放在从 *Fd* 开始的向量中。

对于多操作数指令，*Op* 作用于 *Fm* 中的单个值，也作用于从 *Fn* 开始的向量中的值。结果存放在从 *Fd* 开始的向量中。

5.15.4 VFP 指令和向量记号

本节仅适用于 `armasm`。C 和 C++ 汇编程序中的嵌入式汇编程序不接受这些指令或向量记号。

在新代码中已弃用 VFP 向量模式，并且统一汇编语言不支持向量记号。若要使用向量记号，必须使用旧的 VFP 助记符。有关详细信息，请参阅第5-102 页的 *UAL VFP 预助记符*。旧 VFP 助记符和 UAL VFP 助记符可以混合使用。

您可以在代码中做出关于 VFP 向量长度和跨距的断言，并让汇编程序检查它们。请参阅:

- 第5-105 页的*VFPASSERT SCALAR*
- 第5-106 页的*VFPASSERT VECTOR*。

如果使用 `VFPASSERT` 指令，则必须在用旧助记符编写的所有 VFP 数据处理指令中指定向量详细信息。第5-104 页的*向量记号*中对向量记号进行了说明。如果未使用 `VFPASSERT` 指令，则不能使用此记号。

UAL VFP 预助记符

在 UAL 助记符使用 .F32 指令单精度数据的位置，UAL 预助记符使用 S 追加到指令助记符。例如：VABS.32 为 FABSS。

在 UAL 助记符使用 .F64 指定双精度数据时，UAL 预助记符使用 D 追加到该指令助记符。例如：VCMPE.64 为 FCMPED。

表5-14 显示了受 VFP 向量模式影响的指令的 UAL 预助记符。无论 LEN 和 STRIDE 的设置如何，其他所有 VFP 指令均始终为标量。

表5-14 UAL VFP 预助记符

UAL 助记符	等效的 UAL 预助记符
VABS	FABS
VADD	FADD
VMOV（立即数）	FCONST [a]
VMOV（寄存器）	FCPY
VDIV	FDIV
VMLA	FMAC
VNMLS	FMSC
VMUL	FMUL
VNEG	FNEG
VMLS	FNMAC
VNMLA	FNMSC
VNMUL	FNMUL
VSQRT	FSQRT
VSUB	FSUB

a. VMOV（立即数）中的立即数是要加载的浮点数。FCONST 中的立即数是在生成要加载的浮点数的指令中编码的数字。有关详细信息，请参阅第5-103 页的*FCONST 中的立即值*。

FCONST 中的立即值

表5-15 显示了可使用 `FCONST` 加载的浮点常数。为了清楚起见，将省略尾零。必须包含在 `FCONST` 指令中的立即值是二进制数 `abcdefgh` 的十进制表示形式，其中：

`a` 为 0 或 1；前者表示正数，后者表示负数。

`bcd` 显示在列标题中

`efgh` 显示在行标题中。

也可以使用后跟十六进制表示形式的 `0x`。

表5-15 浮点常数值

bcd	000	001	010	011	100	101	110	111
efgh								
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

向量记号

在 VFP 预数据处理指令中，使用尖括号指定 VFP 寄存器向量：

- `s`*n* 是一个单精度标量寄存器 *n*
- `s`*n*`<>` 是一个单精度向量，其长度和跨距由当前向量长度和跨距给出，从寄存器 *n* 开始
- `s`*n*`<`*L*`>` 是一个单精度向量，长度为 L，跨距为 1，从寄存器 *n* 开始
- `s`*n*`<`*L*`:`*S*`>` 是一个单精度向量，长度为 *L*，跨距为 *S*，从寄存器 *n* 开始
- `d`*n* 是一个双精度标量寄存器 *n*
- `d`*n*`<>` 是一个双精度向量，其长度和跨距由当前向量长度和跨距给出，从寄存器 *n* 开始
- `d`*n*`<`*L*`>` 是一个双精度向量，长度为 L，跨距为 1，从寄存器 *n* 开始
- `d`*n*`<`*L*`:`*S*`>` 是一个双精度向量，长度为 *L*，跨距为 *S*，从寄存器 *n* 开始。

可以将此向量记号与用 `DN` 和 `SN` 指令定义的名称一起使用（请参阅第7-15 页的 *QN、DN 和 SN*）。

不能在 `DN` 和 `SN` 指令中单独使用此向量记号。

VFPASSERT SCALAR

`VFPASSERT SCALAR` 指令通知汇编程序下列 VFP 指令处于标量模式。

语法

```
VFPASSERT SCALAR
```

用法

使用 `VFPASSERT SCALAR` 指令标记 VFP 模式是 `VECTOR` 的任何代码块的结尾。

将 `VFPASSERT SCALAR` 指令放在紧靠发生变化的指令的后面。这通常是 `FMXR` 指令，但也可以是 `BL` 指令。

如果某个函数希望在退出时使 VFP 处于向量模式，可以在紧靠最后一个指令的后面放一个 `VFPASSERT SCALAR` 指令。此类函数不符合 AAPCS。有关详细信息，请参阅 *install_directory*\Documentation\Specifications\... 中的*《ARM 体系结构的过程调用标准》*规范 aapcs.pdf。

另请参阅:

- 第5-104 页的*向量记号*
- 第5-106 页的*VFPASSERT VECTOR*。

---注意---

此指令不会生成任何代码。它只是程序员所做的断言。如果任何此类断言彼此不一致，或者与 VFP 数据处理指令中的任何向量记号不一致，那么汇编程序会生成错误消息。

汇编程序将 `VFPASSERT SCALAR` 指令后面的 VFP 数据处理指令中的向量记号视为错误，即使该向量的长度为 1 也是如此。

示例

```
    VFPASSERT   SCALAR               ; scalar mode
    faddd       d4, d4, d0           ; okay
    fadds       s4<3>, s0, s8<3>     ; ERROR, vector in scalar mode
    fabss       s24<1>, s28<1>       ; ERROR, vector in scalar mode
                                     ; (even though length==1)
```


VFPASSERT VECTOR

VFPASSERT VECTOR 指令通知汇编程序下列 VFP 指令处于向量模式。它也可指定向量的长度和跨距。

语法

VFPASSERT VECTOR[<[*n*[:*s*]]>]

其中:

n 是向量长度 1 到 8。

s 是向量跨距 1 到 2。

用法

使用 VFPASSERT VECTOR 指令来标记 VFP 模式为 VECTOR 的指令块的开始，并标记向量的长度和跨距的变化。

将 VFPASSERT VECTOR 指令紧靠放在发生变化的指令的后面。这通常是 FMXR 指令，但也可以是 BL 指令。

如果某一函数希望在输入时使 VFP 处于向量模式，可以在紧靠第一个指令的前面放一个 VFPASSERT VECTOR 指令。此类函数不符合 AAPCS。有关详细信息，请参阅 *install_directory*\Documentation\Specifications\... 中的*《ARM 体系结构的过程调用标准》*规范 aapcs.pdf。

请参阅:

- 第5-104 页的*向量记号*
- 第5-105 页的*VFPASSERT SCALAR*。

——注意——

此指令不会生成任何代码。它只是程序员所做的断言。如果任何此类断言彼此不一致，或者与 VFP 数据处理指令中的任何向量记号不一致，那么汇编程序会生成错误消息。

示例

```
    VMRS    r10,FPSCR          ; UAL mnemonic - could be FMRX instead.
    BIC     r10,r10,#0x00370000
    ORR     r10,r10,#0x00020000    ; set length = 3, stride = 1
    VMSR    FPSCR,r10

    VFPASSERT VECTOR           ; assert vector mode, unspecified length & stride
    faddd  d4, d4, d0          ; ERROR, scalar in vector mode
```



```
fadds  s16<3>, s0, s8<3>   ; okay
fabss  s24<1>, s28<1>      ; wrong length, but not faulted (unspecified)

VMRS    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00030000    ; set length = 4, stride = 1
VMSR    FPSCR,r10

VFPASSERT VECTOR<4>        ; assert vector mode, length 4, stride 1
fadds  s24<4>, s0, s8<4>   ; okay
fabss  s24<2>, s24<2>      ; ERROR, wrong length

VMRS    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00130000    ; set length = 4, stride = 2
VMSR    FPSCR,r10

VFPASSERT VECTOR<4:2>      ; assert vector mode, length 4, stride 2
fadds  s8<4>, s0, s16<4>   ; ERROR, wrong stride
fabss  s16<4:2>, s28<4:2>  ; okay
fadds  s8<>, s2, s16<>     ; okay (s8 and s16 both have
                           ; length 4 and stride 2.
                           ; s2 is scalar.)
```


第 6 章
无线 MMX 技术指令

本章介绍对 ARM® 汇编程序 `armasm` 中的无线 MMX™ 技术指令的支持。它包含以下几节:

6.1 简介

无线 MMX 技术是一组可用于所选 XScale 处理器的*单指令多数据* (SIMD) 指令，可改善一些多媒体应用程序的性能。无线 MMX 技术使用 64 位寄存器，使其能够以一种封装格式操作多个数据元素。

无线 MMX 技术使用 ARM 协处理器 0 和 1 来支持其指令集和数据类型。您可以对使用无线 MMX 技术指令的源代码进行汇编，以在 PXA270 处理器上运行。

无线 MMX 2 技术是无线 MMX 技术的升级版本。

使用 ARM 汇编程序时，请注意:

- 无线 MMX 技术指令仅在指定了支持的处理器时 (`armasm --cpu PXA270`) 才进行汇编。
- PXA270 处理器仅支持用 ARM 或 Thumb® 编写的代码。
- 大部分无线 MMX 技术指令都可以有条件地执行，这取决于 ARM 标记的状态。无线 MMX 技术条件代码与 ARM 条件代码相同。

本章介绍有关 RVCT 中的 ARM 汇编程序提供的无线 MMX 技术支持的信息。本章不提供无线 MMX 技术的详细说明。有关程序员模型的信息和无线 MMX 技术指令集的完整信息，请参阅*《无线 MMX 技术开发指南》*。

6.2 ARM 对无线 MMX 技术的支持

本节介绍了对无线 MMX 和 MMX 2 技术的汇编程序支持。本节介绍了以下内容:

- *寄存器*
- 第6-4 页的*WRN 和 WCN 指令*
- 第6-4 页的*Frame 指令*
- 第6-5 页的*无线 MMX 加载和存储指令*
- 第6-6 页的*无线 MMX 技术和 XScale 指令*。

6.2.1 寄存器

无线 MMX 技术支持两种寄存器类型:

状态和控制寄存器

控制寄存器映射到协处理器 1，并且包括通用寄存器 `wCGR0 - wCGR3` 和 SIMD 标记。有关这些寄存器的详细信息，请参阅表6-1。

使用无线 MMX 技术指令 `TMCR` 和 `TMRC` 可对这些寄存器进行读写操作。

表6-1 状态和控制寄存器

类型	无线 MMX 技术寄存器	CP1 寄存器
协处理器 ID	wCID	c0
控制	wCon	c1
饱和 SIMD 标记	wCSSF	c2
算法 SIMD 标记	wCASF	c3
保留	-	c4 - c7
通用寄存器	wCGR0 - wCGR3	c8 - c11
保留	-	c12 - c15

SIMD 数据寄存器

数据寄存器 (`wR0 - wR15`) 映射到协处理器 0 并且保存 16x64 位数据包。使用无线 MMX 技术伪指令 `TMRRC` 和 `TMCRR` 可在这些寄存器和 ARM 寄存器之间移动数据。

有关寄存器的详细说明，请参阅《*无线 MMX 技术开发指南*》。

汇编无线 MMX 技术指令时，汇编程序接受以下格式的寄存器规范：

- 大小写混合，大小写与无线 MMX 技术规范完全一致，例如 `wR0`、`wCID`、`wCon`
- 全部小写，例如 `wr0`、`wcid`、`wcon`
- 全部大写，例如 `WR0`、`WCID`、`WCON`。

汇编程序支持 `WRN` 和 `WCN` 指令，以指定自己的寄存器名称（请参阅 *WRN 和 WCN 指令*）。

6.2.2 WRN 和 WCN 指令

支持无线 MMX 技术的可用指令：

WCN 为指定的控制寄存器定义一个名称，例如：

```
speed WCN  wcgr0 ; defines speed as a symbol for control reg 0
```

WRN 为指定的 SIMD 数据寄存器定义一个名称，例如：

```
rate WRN  wr6 ; defines rate as a symbol for data reg 6
```

要避免将不同名称用于同一寄存器。请勿使用第3-18 页的*预定义的寄存器和协处理器名称*中列出的任何预定义名称，或第6-3 页的*寄存器*中介绍的寄存器名称。

6.2.3 Frame 指令

无线 MMX 技术寄存器可以按常规方式与 `FRAME` 指令一起使用，以向目标文件添加调试信息（有关详细信息，请参阅第7-40 页的*Frame 指令*）。请注意以下限制：

- 如果您试图将无线 MMX 技术寄存器 `wR0` - `wR9` 或 `wCGR0` - `wCGR3` 推入堆栈中，则会发出一条警告（请参阅第7-44 页的*FRAME PUSH*）。
- 无线 MMX 技术寄存器不能用作地址偏移量（请参阅第7-42 页的*FRAME ADDRESS* 和第7-48 页的*FRAME RETURN ADDRESS*）。

6.2.4 无线 MMX 加载和存储指令

从/向无线 MMX 协处理器寄存器加载和存储字节、半字、字或双字。

语法

```
op<type>{cond} wRd, [Rn {, #{-}offset}]{!}
op<type>{cond} wRd, [Rn], #{-}offset
opW{cond} wRd, label
opD{cond} wRd, label
opD wRd, [Rn], {-}Rm {, LSL #imm4}]{!}      ; MMX2 only
opD wRd, [Rn], {-}Rm {, LSL #imm4}          ; MMX2 only
opW wCd, [Rn {, #{-}offset}]{!}
opW wCd, [Rn], #{-}offset
```

其中:

op 可为以下指令之一:

WLDR 加载无线 MMX 寄存器

WSTR 存储无线 MMX 寄存器。

<type> 是下列项之一:

B 字节

H 半字

W 字

D 双字。

cond 是一个可选的条件代码（请参阅第2-17 页的*条件执行*）。

wRd 是要加载或保存的无线 MMX 寄存器。

Rn 是内存地址所基于的寄存器。

offset 是直接偏移量。如果省略了偏移量，则该指令为零偏移指令。

! 是一个可选的后缀。如果有 !，则该指令为前变址指令。

label 是一个程序相对的表达式。有关详细信息，请参阅第3-32 页的*相对寄存器和程序相对的表达式*。

label 必须位于当前指令的 +/- 1020 字节范围内。

Rm 是一个寄存器，包含要用作偏移量的值。*Rm* 不能为 r15。

imm4 包含要将 *Rm* 左移的位数，取值范围为 0-15。

将常数加载到 SIMD 寄存器中

该汇编程序还支持 `WLDRW` 文字加载伪指令，例如：

```
WLDRW wr0, =0x114
```

您必须了解：

- 该汇编程序无法加载字节和半字文字。否则会产生降级错误。如果降级，该指令将转换为 `WLDRW`，并生成一个 32 位的文字。这与字节文字加载相同，但是使用的是 32 位字。
- 如果要加载的文字是零，并且目标是一个 SIMD 数据寄存器，则汇编程序将该指令转换为 `WZERO`。
- 非 8 字节对齐的双字加载结果是不可预测的。

6.2.5 无线 MMX 技术和 XScale 指令

无线 MMX 技术指令与 XScale 指令重叠。为了避免冲突，汇编程序具有以下限制：

- 不能在同一汇编中混合使用 XScale 指令与无线 MMX 技术指令。
- 无线 MMX 技术 `TMIA` 指令有一个与 XScale `MIA` 指令重叠的 `MIA` 助记符。您必须了解：
  - `MIA acc0, *Rm*, *Rs*` 在 XScale 中可接受，但是在无线 MMX 技术中会出错。
  - `MIA wR0, *Rm*, *Rs*` 和 `TMIA wR0, *Rm*, *Rs*` 在无线 MMX 技术中可接受。
  - `TMIA acc0, *Rm*, *Rs*` 在 XScale 中会出错（XScale 没有 `TMIA` 指令）。

有关 XScale 指令的详细信息，请参阅第4-128 页的*其他指令*。

6.3 无线 MMX 指令

表6-2 列出了无线 MMX 技术指令集。可使用该表查找*《无线 MMX 技术开发指南》*中所介绍的各个指令。另请参阅伪指令（第6-9 页的表6-3）。

在本节中，无线 MMX 技术寄存器由 *wRn*、*wRd* 表示，ARM 寄存器表示为 *Rn*、*Rd*。

表6-2 无线 MMX 技术指令

助记符	示例
`TANDC`	`TANDCB r15`
`TBCST`	`TBCSTB wr15, r1`
`TEXTRC`	`TEXTRCB r15, #0`
`TEXTRM`	`TEXTRMUBCS r3, wr7, #7`
`TINSR`	`TINSRB wr6, r11, #0`

`TMIA,`
`TMIAPH,`
`TMIAxy`
	`TMIANE wr1, r2, r3`
	`TMIAPH wr4, r5, r6`
	`TMIABB wr4, r5, r6`
	`MIAPHNE wr4, r5, r6`

`TMOVMSK`	`TMOVMSKBNE r14, wr15`
`TORC`	`TORCB r15`
`WACC`	`WACCBGE wr1, wr2`
`WADD`	`WADDBGE wr1, wr2, wr13`
`WALIGNI, WALIGNR`	`WALIGNI wr7, wr6, wr5,#3` `WALIGNR0 wr4, wr8, wr12`
`WAND, WANDN`	`WAND wr1, wr2, wr3` `WANDN wr5, wr5, wr9`
`WAVG2`	`WAVG2B wr3, wr6, wr9` `WAVG2BR wr4, wr7, wr10`
`WCMPEQ`	`WCMPEQB wr0, wr4, wr2`
`WCMPGT`	`WCMPGTUB wr0, wr4, wr2`
`WLDR`	`WLDRB wr1, [r2, #0]`
`WMAC`	`WMACU wr3, wr4, wr5`

表6-2 无线 MMX 技术指令 (续)

助记符	示例
WMADD	WMADDU wr3, wr4, wr5
WMAX, WMIN	WMAXUB wr0, wr4, wr2 WMINSB wr0, wr4, wr2
WMUL	WMULUL wr4, wr2, wr3
WOR	WOR wr3, wr1, wr4
WPACK	WPACKHUS wr2, wr7, wr1
WROR	WRORH wr3, wr1, wr4
WSAD	WSADB wr3, wr5, wr8
WSHUFH	WSHUFH wr8, wr15, #17
WSLL, WSRL	WSLLH wr3, wr1, wr4 WSRLHG wr3, wr1, wcgr0
WSRA	WSRAH wr3, wr1, wr4 WSRAHG wr3, wr1, wcgr0
WSTR	WSTRB wr1, [r2, #0] WSTRW wc1, [r2, #0]
WSUB	WSUBBGE wr1, wr2, wr13
WUNPCKEH, WUNPCKEL	WUNPCKEHUB wr0, wr4 WUNPCKELSB wr0, wr4
WUNPCKIH, WUNPCKIL	WUNPCKIHB wr0, wr4, wr2 WUNPCKILH wr1, wr5, wr3
WXOR	WXOR wr3, wr1, wr4

6.3.1 伪指令

表6-3 提供了无线 MMX 技术伪指令的概述。可使用该表查找*《无线 MMX 技术开发指南》*和第 4 章 *ARM 和 Thumb 指令*中所介绍的指令。

表6-3 无线 MMX 技术伪指令

助记符	简单说明	示例
TMCR	将源寄存器 *Rn* 的内容移到控制寄存器 *wCn* 中。 映射到 ARM MCR 协处理器指令（页4-122）。	TMCR wc1, r10
TMCRR	将两个源寄存器 *RnLo* 和 *RnHi* 的内容移到目标寄存器 *wRd*。不要将 r15 用于 *RnLo* 或 *RnHi*。映射到 ARM MCRR 协处理器指令（页4-122）。	TMCRR wr4, r5, r6
TMRC	将控制寄存器 *wCn* 的内容移到目标寄存器 *Rd*。不要将 r15 用于 *Rd*。映射到 ARM MRC 协处理器指令（页4-124）。	TMRC r1, wc2
TMRRC	将源寄存器 *wRn* 的内容移到两个目标寄存器 *RdLo* 和 *RdHi*。不要将 r15 用于其中任一目标寄存器。*RdLo* 和 *RdHi* 必须为不同的寄存器，否则结果将不可预测。 映射到 ARM MRRC 协处理器指令（页4-124）。	TMRRC r1, r0, wr2
WMOV	将源寄存器 *wRn* 的内容移到目标寄存器 *wRd*。此指令是 WOR 的一种形式（请参阅第6-7 页的表6-2）。	WMOV wr1, wr8
WZERO	清零目标寄存器 *wRd*。此指令是 WANDN 的一种形式（请参阅第6-7 页的表6-2）。	WZERO wr1

第 7 章
指令参考

本章介绍 ARM® 汇编程序 `armasm` 提供的指令。它包含以下几节:

——注意——

这些指令在 ARM C 和 C++ 编译器的内联汇编程序中不可用。

7.1 按字母顺序排列的指令表

表7-1 显示了指令的完整列表。可以使用该表查找本章后面介绍的各个指令。

表7-1 指令的位置

指令	页码	指令	页码	指令	页码
ALIGN	页7-63	EXPORT *或* GLOBAL	页7-70	LTORG	页7-18
ARM *和* CODE32	页7-61	EXPORTAS	页7-72	MACRO *和* MEND	页7-32
AREA	页7-65	EXTERN	页7-74	MAP	页7-19
ASSERT	页7-55	FIELD	页7-20	MEND *请参阅* MACRO	页7-32
CN	页7-13	FRAME ADDRESS	页7-42	MEXIT	页7-35
CODE16	页7-61	FRAME POP	页7-43	NOFP	页7-78
COMMON	页7-30	FRAME PUSH	页7-44	OPT	页7-57
CP	页7-14	FRAME REGISTER	页7-46	PRESERVE8 *请参阅* REQUIRE8	页7-79
DATA	页7-30	FRAME RESTORE	页7-47	PROC *请参阅* FUNCTION	页7-53
DCB	页7-22	FRAME SAVE	页7-49	QN	页7-15
DCD *和* DCDU	页7-23	FRAME STATE REMEMBER	页7-50	RELOC	页7-9
DCDO	页7-24	FRAME STATE RESTORE	页7-51	REQUIRE	页7-78
DCFD *和* DCFDU	页7-25	FRAME UNWIND ON *或* OFF	页7-52	REQUIRE8 *和* PRESERVE8	页7-79
DCFS *和* DCFSU	页7-26	FUNCTION *或* PROC	页7-53	RLIST	页7-12
DCI	页7-27	GBLA、GBLL *和* GBLS	页7-5	RN	页7-11
DCQ *和* DCQU	页7-28	GET *或* INCLUDE	页7-73	ROUT	页7-80
DCW *和* DCWU	页7-29	GLOBAL *请参阅* EXPORT	页7-70	SETA、SETL *和* SETS	页7-8
DN	页7-15	IF、ELSE、ENDIF *和* ELIF	页7-36	SN	页7-15
ELIF、ELSE *请参阅* IF	页7-36	IMPORT	页7-74	SPACE	页7-21
END	页7-68	INCBIN	页7-76	SUBT	页7-59
ENDFUNC *或* ENDP	页7-54	INCLUDE *请参阅* GET	页7-73	THUMB	页7-61

表7-1 指令的位置 (续)

指令	页码	指令	页码	指令	页码
ENDIF *请参阅* IF	页7-36	INFO	页7-56	THUMBX	页7-61
ENTRY	页7-68	KEEP	页7-77	TTL	页7-59
EQU	页7-69	LCLA、LCLL *和* LCLS	页7-7	WHILE *和* WEND	页7-39

7.2 符号定义指令

本节介绍下列指令:

- 第7-5 页的*GBLA、GBLL 和 GBLS*
  声明全局算术、逻辑"或"字符串变量。
- 第7-7 页的*LCLA、LCLL 和 LCLS*
  声明局部算术、逻辑"或"字符串变量。
- 第7-8 页的*SETA、SETL 和 SETS*
  设置算术、逻辑"或"字符串变量的值。
- 第7-9 页的*RELOC*
  在目标文件中对 ELF 重定位进行编码。
- 第7-11 页的*RN*
  定义指定寄存器的名称。
- 第7-12 页的*RLIST*
  为一组通用寄存器定义一个名称。
- 第7-13 页的*CN*
  定义协处理器寄存器名称。
- 第7-14 页的*CP*
  定义协处理器名称。
- 第7-15 页的*QN、DN 和 SN*
  定义双精度或单精度 VFP 寄存器名称。

7.2.1 GBLA、GBLL 和 GBLS

GBLA 指令声明一个全局算术变量，并将其值初始化为 0。

GBLL 指令声明一个全局逻辑变量，并将其值初始化为 {FALSE}。

GBLS 指令声明一个全局字符串变量，并将其值初始化为空字符串 ""。

语法

<gblx> variable

其中:

<gblx> 是 GBLA、GBLL 或 GBLS 之一。

variable 是变量的名称。*variable* 在一个源文件内的符号中必须是唯一的。

用法

对于已经定义的变量，使用上述指令之一，可将其重新初始化为上面给出的值。

变量的作用域以其所在的源文件为限。

用 SETA、SETL 或 SETS 指令来设置变量的值（请参阅第7-8 页的*SETA、SETL 和 SETS*）。

有关声明局部变量的信息，请参阅第7-7 页的*LCLA、LCLL 和 LCLS*。

全局变量也可以用 -predefine 汇编程序命令行选项来设置。有关详细信息，请参阅第3-2 页的*命令语法*。

示例

示例 7-1 声明变量 `objectsize`，将 `objectsize` 的值设置为 `0xFF`，然后在后面的 `SPACE` 指令中使用它。

示例 7-1

```
            GBLA    objectsize    ; declare the variable name
objectsize  SETA    0xFF          ; set its value
            .
            .                     ; other code
            .
            SPACE   objectsize    ; quote the variable
```

示例 7-2 演示如何在调用 `armasm` 时声明和设置变量。如果要在汇编时设置变量的值，则可以使用此方法。`--pd` 是 `--predefine` 的同义词。

示例 7-2

```
armasm --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```


7.2.2 LCLA、LCLL 和 LCLS

LCLA 指令声明一个局部算术变量，并将其值初始化为 0。

LCLL 指令声明一个局部逻辑变量，并将其值初始化为 {FALSE}。

LCLS 指令声明一个局部字符串变量，并将其值初始化为空字符串 ""。

语法

<lclx> variable

其中:

<lclx> 是 LCLA、LCLL 或 LCLS 之一。

variable 是变量的名称。*variable* 在其所在的宏内必须是唯一的。

用法

对于已经定义的变量，使用上述指令之一，可将其重新初始化为上面给出的值。

变量的作用域以其所在宏的特定实例为限（请参阅第7-32 页的*MACRO 和 MEND*）。

用 SETA、SETL 或 SETS 指令来设置变量的值（请参阅第7-8 页的*SETA*、*SETL 和 SETS*）。

有关声明全局变量的信息，请参阅第7-5 页的*GBLA*、*GBLL 和 GBLS*。

示例

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```


7.2.3 SETA、SETL 和 SETS

SETA 指令用于设置局部或全局算术变量的值。

SETL 指令用于设置局部或全局逻辑变量的值。

SETS 指令用于设置局部或全局字符串变量的值。

语法

variable <setx> expr

其中：

<setx> 是 SETA、SETL 或 SETS 之一。

variable 是由 GBLA、GBLL、GBLS、LCLA、LCLL 或 LCLS 指令声明的变量的名称。

expr 是一个表达式，可以为以下几种类型：

- 对于 SETA，其值是数值型（请参阅第3-29 页的*数字表达式*）
- 对于 SETL，其值是逻辑型（请参阅第3-32 页的*逻辑表达式*）
- 对于 SETS，其值是字符串（请参阅第3-28 页的*字符串表达式*）。

用法

在使用这些指令之前，必须使用全局或局部声明指令声明 *variable*。有关详细信息，请参阅第7-5 页的*GBLA、GBLL 和 GBLS* 和第7-7 页的*LCLA、LCLL 和 LCLS*。

您也可以在命令行上预定义变量名。有关详细信息，请参阅第3-2 页的*命令语法*。

示例

```
                GBLA    VersionNumber
VersionNumber   SETA    21

                GBLL    Debug
Debug           SETL    {TRUE}

                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```


7.2.4 RELOC

RELOC 指令可在目标文件中对 ELF 重定位进行显式编码。

语法

RELOC *n*, *symbol*

RELOC *n*

其中:

n 必须在 0 到 255 之间。

symbol 可以是任意程序相对的标签。

用法

使用 RELOC *n*, *symbol* 可依据 *symbol* 标记的地址创建重定位。

如果在 ARM 或 Thumb 指令后立即使用，RELOC 会在该指令处重定位。如果在 DCB、DCW、DCD 或其他任何数据生成指令后立即使用，则 RELOC 会在数据的开始处重定位。要应用的任何附加代码必须在指令中或者在 DCI 或 DCD 中进行编码。

如果汇编程序已在该位置上发出重定位，则会根据 RELOC 指令中的详细信息更新重定位，例如:

```
DCD     sym2 ; R_ARM_ABS32 to sym32
RELOC   55   ; ... makes it R_ARM_ABS32_NOI
```

在其他所有情况下，RELOC 会出错，例如，在任何非数据生成指令、LTORG、ALIGN 之后，或在 AREA 的最开头。

使用 RELOC *n* 可依据匿名符号（即符号表中的符号 0）创建重定位。如果使用的 RELOC *n* 前面没有汇编程序生成的重定位，则重定位将依据匿名符号。

示例

```
IMPORT  impsym
LDR     r0,[pc,#-8]
RELOC   4, impsym

DCD     0
RELOC   2, sym

DCD     0,1,2,3,4  ; the final word is relocated
RELOC   38,sym2    ; R_ARM_TARGET1
```

7.2.5 RN

`RN` 指令为指定寄存器定义寄存器名称。

语法

name RN *expr*

其中:

name 是要分配给寄存器的名称。*name* 不能与第3-18 页的*预定义的寄存器和协处理器名称*中列出的任何预定义名称相同。

expr 取值为 0 到 15 之间的一个寄存器编号。

用法

使用 `RN` 可为寄存器分配便于记忆的名称，以帮助记忆每个寄存器的用途。要小心避免将不同名称用于同一寄存器。

示例

```
regname     RN  11  ; defines regname for register 11

sqr4        RN  r6  ; defines sqr4 for register 6
```


7.2.6 RLIST

RLIST（寄存器列表）指令为一组通用寄存器指定名称。

语法

name RLIST {*list-of-registers*}

其中：

name 是要为寄存器组指定的名称。*name* 不能与第3-18 页的*预定义的寄存器和协处理器名称*中列出的任何预定义名称相同。

list-of-registers

是一个用逗号分隔的寄存器名称列表和/或寄存器范围。寄存器列表必须括在大括号内。

用法

使用 RLIST 可为要用 LDM 或 STM 指令传送的一组寄存器指定名称。

LDM 和 STM 始终将最低的物理寄存器编号放在内存的最低地址处，而不考虑向 LDM 或 STM 指令提供这些寄存器编号时的顺序。如果您已经定义了自己的符号寄存器名称，则寄存器列表不按递增顺序排列这一点就不太明显。

使用 --diag_warning 1206 汇编程序选项可确保寄存器列表中的寄存器是以递增顺序提供的。如果寄存器不是以递增顺序提供的，则发出一个警告。

示例

```
Context RLIST   {r0-r6,r8,r10-r12,r15}
```

7.2.7 CN

`CN` 指令为协处理器寄存器定义名称。

语法

name `CN` *expr*

其中:

name 是要为协处理器寄存器定义的名称。*name* 不能与第3-18 页的*预定义的寄存器和协处理器名称*中列出的任何预定义名称相同。

expr 取值为 0 到 15 之间的一个协处理器寄存器编号。

用法

使用 `CN` 可为寄存器分配便于记忆的名称，以帮助记忆每个寄存器的用途。

——注意——

要避免将不同名称用于同一寄存器。

从 c0 到 c15 的名称是预先定义的。

示例

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```


7.2.8 CP

CP 指令为指定的协处理器定义名称。协处理器编号必须在 0 到 15 之间。

语法

name CP *expr*

其中:

name 是要分配给协处理器的名称。*name* 不能与第3-18 页的*预定义的寄存器和协处理器名称*中列出的任何预定义名称相同。

expr 取值为 0 到 15 之间的一个协处理器编号。

用法

使用 CP 可为协处理器分配便于记忆的名称，以帮助记忆每个协处理器的用途。

——注意——

要避免将不同名称用于同一协处理器。

---

p0 到 p15 是为 0 到 15 的协处理器预先定义的名称。

示例

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

7.2.9 QN、DN 和 SN

`QN` 指令为指定的 128 位扩展寄存器定义名称。

`DN` 指令为指定的 64 位扩展寄存器定义名称。

`SN` 指令为指定的单精度 VFP 寄存器定义名称。

语法

`name directive expr{.type}{[x]}`

其中:

`directive` 是 `QN`、`DN` 或 `SN`。

`name` 是要分配给扩展寄存器的名称。`name` 不能与第3-18 页的*预定义的寄存器和协处理器名称*中列出的任何预定义名称相同。

`expr` 可以是:

- 一个表达式，对于双精度 VFPv2 寄存器或 NEON 128 位寄存器编号，其取值为 0 到 15 之间，否则为 0 到 31 之间。
- 预定义的寄存器名称，或已在前面指令中定义了的寄存器名称。

`type` 是第5-12 页的*NEON 和 VFP 数据类型*中介绍的任何数据类型。

`[x]`

仅用于 NEON 代码。`[x]` 是指向寄存器的标量索引。

`type` 和 `[x]` 是*扩展表示法*。有关详细信息，请参阅第5-15 页的*扩展记号*，有关用法的示例，请参阅第7-16 页的*扩展表示法示例*。

用法

使用 `QN`、`DN` 或 `SN` 可为扩展寄存器分配便于记忆的名称，以帮助记忆每个扩展寄存器的用途。

——注意——

要避免将不同名称用于同一寄存器。

不能在 `DN` 或 `SN` 指令中指定向量长度（请参阅第5-101 页的*VFP 指令和向量记号*）。

示例

```
energy  DN  6   ; defines energy as a symbol for
                ; VFP double-precision register 6

mass    SN  16  ; defines mass as a symbol for
                ; VFP single-precision register 16
```

扩展表示法示例

```
varA    DN      d1.U16
varB    DN      d2.U16
varC    DN      d3.U16
        VADD    varA,varB,varC      ; VADD.U16 d1,d2,d3
index   DN      d4.U16[0]
result  QN      q5.I32
        VMULL   result,varA,index   ; VMULL.U16 q5,d1,d3[2]
```

7.3 数据定义指令

本节介绍以下用于分配内存、定义数据结构以及设置内存初始值的指令:

- 第7-18 页的*LTORG*
  设置文字池的原点。
- 第7-19 页的*MAP*
  设置存储映射的原点。
- 第7-20 页的*FIELD*
  定义存储映射内的域。
- 第7-21 页的*SPACE*
  分配用零填充的内存块。
- 第7-22 页的*DCB*
  分配内存中的字节，并指定初始内容。
- 第7-23 页的*DCD 和 DCDU*
  分配内存中的字，并指定初始内容。
- 第7-24 页的*DCDO*
  分配内存中的字，并将初始内容指定为相对静态基址寄存器的偏移量。
- 第7-25 页的*DCFD 和 DCFDU*
  分配内存中的双字，并将初始内容指定为双精度浮点数。
- 第7-26 页的*DCFS 和 DCFSU*
  分配内存中的字，并将初始内容指定为单精度浮点数。
- 第7-27 页的*DCI*
  分配内存中的字，并指定初始内容。将位置标记为代码而非数据。
- 第7-28 页的*DCQ 和 DCQU*
  分配内存中的双字，并将初始内容指定为 64 位整数。
- 第7-29 页的*DCW 和 DCWU*
  分配内存中的半字，并指定初始内容。
- 第7-30 页的*COMMON*
  在符号处分配内存块，并指定对齐方式。

- 第7-30 页的*DATA*

  标记代码节内的数据。已不再使用，仅用于提供向下兼容性。

7.3.1 LTORG

LTORG 指令指示汇编程序立即汇编当前文字池。

语法

LTORG

用法

汇编程序在每个代码节结尾处汇编当前文字池。代码节的结束位置由后续节开始处的 AREA 指令确定，或由汇编代码的结束位置确定。

这些缺省文字池有时会超出某些 LDR、FLDD 和 FLDS 伪指令的范围。请参阅第4-153 页的*LDR 伪指令*。使用 LTORG 可确保在指定范围内汇编文字池。大型程序可能需要多个文字池。

将 LTORG 指令放在无条件跳转或子例程返回指令之后，以使处理器不会试图将常数作为指令来执行。

汇编程序对文字池中的数据进行字对齐。

示例

```
        AREA    Example, CODE, READONLY
start   BL      func1

func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.

data    SPACE   4200            ; Clears 4200 bytes of memory,
                                ; starting at current location.
        END                     ; Default literal pool is empty.
```

7.3.2 MAP

MAP 指令将存储映射的原点设置为指定的地址。存储映射位置计数器 {VAR} 也被设置为相同的地址。^ 是 MAP 的同义词。

语法

MAP *expr*{,*base-register*}

其中:

expr 是一个数值表达式或程序相对的表达式:

- 如果未指定 *base-register*，则 *expr* 的取值为存储映射的起始地址。存储映射位置计数器也设置为此地址。
- 如果 *expr* 是程序相对的，则在映射中使用标签之前，必须已定义该标签。在第一轮汇编时，映射需要标签的定义。

base-register

指定一个寄存器。如果指定了 *base-register*，则存储映射的起始地址是 *expr* 与运行时的 *base-register* 值之和。

用法

将 MAP 指令与 FIELD 指令配合使用可描述存储映射。

指定 *base-register* 可定义寄存器相对的标签。在由后面的 FIELD 指令定义的所有标签中，基址寄存器变成隐含的，直到出现下一个 MAP 指令。寄存器相对的标签可用在加载和存储指令中。有关示例，请参阅第7-20 页的*FIELD*。

可以任意多次使用 MAP 指令，以定义多个存储映射。

在使用第一个 MAP 指令之前，{VAR} 计数器被设置为 0。

示例

```
        MAP     0,r9
        MAP     0xff,r9
```


7.3.3 FIELD

FIELD 指令描述已使用 MAP 指令定义的存储映射内的空间。# 是 FIELD 的同义词。

语法

{*label*} FIELD *expr*

其中：

label 是一个可选的标签。如果指定该项，则将存储位置计数器 {VAR} 的值分配给 *label*。然后存储位置计数器增加大小为 *expr* 的值。

expr 是一个表达式，取值为存储计数器将增大的字节数。

用法

如果存储映射是由指定 *base-register* 的 MAP 指令设置的，则在由后续 FIELD 指令定义的所有标签中，基址寄存器是隐含的，直到出现下一个 MAP 指令为止。在加载和存储指令中可引用这些寄存器相对的标签（请参阅第7-19 页的*MAP*）。

示例

以下示例演示如何使用 MAP 和 FIELD 指令定义寄存器相对的标签。

```
    MAP     0,r9        ; set {VAR} to the address stored in r9
    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [r9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```


7.3.4 SPACE

`SPACE` 指令保留一个用零填充的内存块。`%` 是 `SPACE` 的同义词。

语法

{*label*} SPACE *expr*

其中:

expr 取值为要保留的填零字节数（请参阅第3-29 页的*数字表达式*）。

用法

使用 `ALIGN` 指令可对齐 `SPACE` 指令后的任何代码。有关详细信息，请参阅第7-63 页的*ALIGN*。

另请参阅:

- 第7-22 页的*DCB*
- 第7-23 页的*DCD 和 DCDU*
- 第7-24 页的*DCDO*
- 第7-29 页的*DCW 和 DCWU*。

示例

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255     ; defines 255 bytes of zeroed store
```


7.3.5 DCB

DCB 指令可分配一个或多个字节的内存，并定义内存的运行时初值。= 是 DCB 的同义词。

语法

```
{label} DCB expr{,expr}...
```

其中:

expr 可以是:

- 一个数值表达式，取值为 -128 到 255 之间的一个整数（请参阅第3-29 页的*数字表达式*）。
- 用引号括起来的字符串。字符串的字符加载到内存的连续字节中。

用法

如果 DCB 后面有一个指令，使用 ALIGN 指令可确保该指令是对齐的。有关详细信息，请参阅第7-63 页的*ALIGN*。

另请参阅:

- 第7-23 页的*DCD 和 DCDU*
- 第7-28 页的*DCQ 和 DCQU*
- 第7-29 页的*DCW 和 DCWU*
- 第7-21 页的*SPACE*。

示例

与 C 语言字符串不同的是，ARM 汇编程序字符串不是空终止的。可以使用 DCB，按如下方式构造空终止的 C 字符串:

```
C_string   DCB  "C_string",0
```

7.3.6 DCD 和 DCDU

DCD 指令可分配一个或多个字的内存，在四个字节的边界上对齐，并定义内存的运行时初值。

& 是 DCD 的同义词。

DCDU 与之相同，不过内存对齐是任意的。

语法

{*label*} DCD{U} *expr*{,*expr*}

其中:

expr 可以是:
- 一个数值表达式（请参阅第3-29 页的*数字表达式*）。
- 一个程序相对的表达式。

用法

必要时，DCD 可在定义的第一个字前最多插入三个填充字节，以实现四字节对齐。

如果不需要对齐，则可使用 DCDU。

另请参阅:
- 第7-22 页的*DCB*
- 第7-29 页的*DCW 和 DCWU*
- 第7-28 页的*DCQ 和 DCQU*
- 第7-21 页的*SPACE*。

示例

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20

data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06

        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

7.3.7 DCDO

`DCDO` 指令可分配一个或多个字的内存，在四个字节的边界上对齐，并将内存的运行时初值定义为相对*静态基址寄存器* sb (r9) 的偏移量。

语法

{*label*} DCDO *expr*{,*expr*}...

其中：

expr 是一个寄存器相对的表达式或标签。基址寄存器必须是 sb。

用法

使用 `DCDO` 可为静态基址寄存器的相对浮动地址分配内存空间。

示例

```
        IMPORT  externsym
        DCDO    externsym   ; 32-bit word relocated by offset of
                            ; externsym from base of SB section.
```


7.3.8 DCFD 和 DCFDU

DCFD 指令为字对齐的双精度浮点值分配内存，并定义内存的运行时初值。双精度数占用两个字，并且必须字对齐才能用在算术运算中。

DCFDU 与之相同，不过内存对齐是任意的。

语法

```
{label} DCFD{U} fpliteral{,fpliteral}...
```

其中:

fpliteral 是一个双精度浮点文字（请参阅第3-31 页的*浮点文本*）。

用法

必要时，汇编程序可在定义的第一个数字前最多插入三个填充字节，以实现四字节对齐。

如果不需要对齐，则可使用 DCFDU。

在将 *fpliteral* 转换为内部形式时，所使用的字顺序是由选定的浮点体系结构来控制的。如果选择了 --fpu none 选项，则不能使用 DCFD 或 DCFDU。

双精度数值的范围是:

- 最大值 1.79769313486231571e+308
- 最小值 2.22507385850720138e-308。

另请参阅第7-26 页的*DCFS 和 DCFSU*。

示例

```
        DCFD    1E308,-4E-100
        DCFDU   10000,-.1,3.1E26
```


7.3.9 DCFS 和 DCFSU

DCFS 指令为字对齐的单精度浮点数分配内存，并定义内存的运行时初值。单精度数占用一个字，并且必须字对齐才能用在算术运算中。

DCDSU 与之相同，不过内存对齐是任意的。

语法

{*label*} DCFS{U} *fpliteral*{,*fpliteral*}...

其中:

fpliteral 是一个单精度浮点文字（请参阅第3-31 页的*浮点文本*）。

用法

必要时，DCFS 可在定义的第一个数前最多插入三个填充字节，以实现四字节对齐。

如果不需要对齐，则可使用 DCFSU。

单精度数值的范围是:

- 最大值 3.40282347e+38
- 最小值 1.17549435e-38。

另请参阅第7-25 页的*DCFD 和 DCFDU*。

示例

```
        DCFS    1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```


7.3.10 DCI

在 ARM 代码中，`DCI` 指令分配一个或多个字的内存，在四个字节的边界上对齐，并定义内存的运行时初值。

在 Thumb 代码中，`DCI` 指令分配一个或多个半字的内存，在两个字节的边界上对齐，并定义内存的运行时初值。

语法

`{label} DCI{.W} expr{,expr}`

其中:

expr 是一个数值表达式（请参阅第3-29 页的*数字表达式*）。

.W 如果有，表示必须在 Thumb 代码中插入四个字节。

用法

`DCI` 指令与 `DCD` 或 `DCW` 指令非常相似，但位置被标记为代码而不是数据。在为当前汇编程序版本不支持的新指令编写宏时，可使用 `DCI`。

在 ARM 代码中，必要时，`DCI` 会在定义的第一个字前最多插入三个填充字节，以实现四字节对齐。在 Thumb 代码中，`DCI` 在必要时会插入一个初始填充字节，以实现两个字节的对齐。

可以使用 `DCI` 将位模式插入指令流中。例如，使用:

```
DCI 0x46c0
```

插入 Thumb 操作 `MOV r8,r8`。

另请参阅第7-23 页的*DCD 和 DCDU* 和第7-29 页的*DCW 和 DCWU*。

宏示例

```
    MACRO           ; this macro translates newinstr Rd,Rm
                    ; to the appropriate machine code
    newinst     $Rd,$Rm
    DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
    MEND
```

Thumb-2 示例

```
    DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```


7.3.11 DCQ 和 DCQU

DCQ 指令可分配一个或多个八字节的内存块，在四字节的边界上对齐，并定义内存的运行时初值。

DCQU 与之相同，不过内存对齐是任意的。

语法

```
{label} DCQ{U} {-}literal{,{-}literal}...
```

其中:

literal 是一个 64 位数字文字（请参阅第3-30 页的*数字文本*）。

允许的数值范围是 0 到 2^{64}-1。

除了数字文字中通常允许的字符外，您还可以在 *literal* 的前面缀以减号。在这种情况下，允许的数值范围是 -2^{63} 到 -1。

指定 -*n* 的结果与指定 2^{64}-*n* 的结果相同。

用法

必要时，DCQ 可在定义的第一个八字节块前最多插入三个填充字节，以实现四字节对齐。

如果不需要对齐，则可使用 DCQU。

另请参阅:

- 第7-22 页的*DCB*
- 第7-23 页的*DCD 和 DCDU*
- 第7-29 页的*DCW 和 DCWU*
- 第7-21 页的*SPACE*。

示例

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101     ; 2_101 means binary 101.
        DCQU    number+4       ; number must already be defined.
```


7.3.12 DCW 和 DCWU

`DCW` 指令可分配一个或多个半字的内存，在两个字节的边界上对齐，并定义内存的运行时初值。

`DCWU` 与之相同，不过内存对齐是任意的。

语法

`{`*`label`*`} DCW{U}` *`expr`*`{,`*`expr`*`}...`

其中:

expr 是一个数值表达式，取值为 -32768 到 65535 之间的一个整数（请参阅第3-29 页的*数字表达式*）。

用法

必要时，`DCW` 可在定义的第一个半字前插入一个填充字节，以实现两个字节的对齐。

如果不需要对齐，则可使用 `DCWU`。

另请参阅:

- 第7-22 页的*DCB*
- 第7-23 页的*DCD 和 DCDU*
- 第7-28 页的*DCQ 和 DCQU*
- 第7-21 页的*SPACE*。

示例

```
data    DCW     -225,2*number   ; number must already be defined

        DCWU    number+4
```


7.3.13 COMMON

COMMON 指令在指定符号处分配所定义大小的内存块。您还可以指定内存的对齐方式。如果省略了对齐方式，则缺省的对齐方式为 4。如果省略大小，则缺省大小为 0。

可如同访问其他内存一样访问此内存，不过不在目标文件中分配任何空间。

语法

COMMON *symbol*{,*size*{,*alignment*}}

其中:

symbol 是符号的名称。符号名区分大小写。

size 是要保留的字节数。

alignment 是对齐方式。

用法

链接器在链接阶段会将所需空间分配为初始值为零的内存。

示例

```
        COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

7.3.14 DATA

DATA 指令不再需要。汇编程序会忽略它。

7.4 汇编控制指令

本节介绍的以下指令用于控制条件汇编、循环、包含以及宏:

- 第7-32 页的*MACRO 和 MEND*
- 第7-35 页的*MEXIT*
- 第7-36 页的*IF*、*ELSE*、*ENDIF 和 ELIF*
- 第7-39 页的*WHILE 和 WEND*。

7.4.1 嵌套指令

下列结构可以嵌套到 256 层的总深度:

- `MACRO` 定义
- `WHILE...WEND` 循环
- `IF...ELSE...ENDIF` 条件结构
- `INCLUDE` 文件包含。

该限制适用于放到一起的所有结构，不论它们是如何嵌套的。并不是每种结构类型都是 256 层限制。

7.4.2 MACRO 和 MEND

MACRO 指令标记一个宏定义的开始。宏扩展在 MEND 指令处终止。有关详细信息，请参阅第2-45 页的*使用宏*。

语法

有两个指令用于定义一个宏。其语法是:

```
        MACRO
{$label} macroname{$cond} {$parameter{,$parameter}...}
        ; code
        MEND
```

其中:

$label 是由调用宏时提供的符号替换的参数。该符号通常是一个标签。

macroname 是宏的名称。它不能以指令或命令名开始。

$cond 是专用于包含条件代码的特殊参数。允许有效条件代码以外的值。

$parameter 是调用宏时被替换的一个参数。可用以下格式设置参数的缺省值:

$parameter="default value"

如果缺省值内或两端有空格，则必须使用双引号。

用法

如果在一个宏内开始任何 WHILE...WEND 循环或 IF...ENDIF 条件，则必须在到达 MEND 指令之前结束它们。如果要允许提前从宏内退出（例如从一个循环内退出），请参阅第7-35 页的*MEXIT*。

在宏体内，可以像其他变量一样使用类似于 $*label*、$*parameter* 或 $*cond* 等参数（请参阅第3-23 页的*汇编时的变量替换*）。每次调用宏时，都会为它们指定新值。参数必须以 $ 开始，以区别于常规符号。可以使用任意数目的参数。

$label 是可选的。如果宏定义了内部标签，则它很有用。它被当作宏的一个参数。它不一定代表宏扩展中的第一个指令。宏定义任何标签的位置。

使用 | 作为自变量来使用参数的缺省值。如果省略该自变量，则使用一个空字符串。

在使用多个内部标签的宏内，将每个内部标签定义为带有不同后缀的基址标签会很有用。

如果在扩展中不需要空格，则在参数与后面的文本或参数之间使用一个圆点。不得在参数与前面的文本之间使用圆点。

您可对条件代码使用 `$cond` 参数。使用一元运算符 `:REVERSE_CC:` 可得到条件代码的取反代码，使用 `:CC_ENCODING:` 可得到条件代码的 4 位编码。

宏可定义局部变量的作用域（请参阅第7-7 页的*LCLA、LCLL 和 LCLS*）。

宏可以嵌套（请参阅第7-31 页的*嵌套指令*）。

示例

```
 ; macro definition

                MACRO                       ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                        ; end macro definition

 ; macro invocation

abc             xmac    subr1,de            ; invoke macro
                ; code                      ; this is what is
abcloop1        ; code                      ; is produced when
                ; code                      ; the xmac macro is
                BGE     abcloop1            ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

使用宏来生成汇编时的诊断信息：

```
        MACRO                          ; Macro definition
        diagnose  $param1="default"    ; This macro produces
        INFO      0,"$param1"          ; assembly-time diagnostics
        MEND                           ; (on second assembly pass)
```

```
 ; macro expansion

        diagnose            ; Prints blank line at assembly-time
        diagnose "hello"    ; Prints "hello" at assembly-time
        diagnose |          ; Prints "default" at assembly-time
```

条件宏示例

```
        AREA    codx, CODE, READONLY

 ; macro definition

        MACRO
        Return$cond
        [ {ARCHITECTURE} <> "4"
        BX$cond  lr
        |
        MOV$cond  pc,lr
        ]
        MEND

 ; macro invocation

fun     PROC
        CMP      r0,#0
        MOVEQ    r0,#1
        ReturnEQ
        MOV      r0,#0
        Return
        ENDP

        END
```


7.4.3 MEXIT

MEXIT 指令用于在宏结束之前退出宏定义。

用法

当需要从一个宏体内退出时，可使用 MEXIT。在退出该宏之前，宏体内任何未结束的 WHILE...WEND 循环或 IF...ENDIF 条件将由汇编程序来结束。

另请参阅第7-32 页的*MACRO 和 MEND*。

示例

```
        MACRO
$abc    example abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```


7.4.4 IF、ELSE、ENDIF 和 ELIF

IF 指令引入一个条件，用于决定是否汇编一个指令和/或命令序列。[是 IF 的同义词。

ELSE 指令标记指令和/或命令序列的开始，此序列为在不满足前面的条件时要汇编的序列。| 是 ELSE 的同义词。

ENDIF 指令标记要进行条件汇编的指令和/或命令序列的结束。] 是 ENDIF 的同义词。

ELIF 指令生成一个与 ELSE IF 等效的结构，无需嵌套或重复条件。有关详细信息，请参阅第7-37 页的*使用 ELIF*。

语法

```
IF logical-expression      ...    {ELSE      ...}    ENDIF
```

其中:

logical-expression

是一个取值为 {TRUE} 或 {FALSE} 的表达式。

请参阅第3-39 页的*关系运算符*。

用法

对于只能在指定条件下被汇编或执行的指令和/或命令序列，可使用 IF 和 ENDIF 以及可选的 ELSE。

IF...ENDIF 条件可以嵌套（请参阅第7-31 页的*嵌套指令*）。

使用 ELIF

不使用 ELIF 时，可以构造类似如下的一组嵌套的条件指令:

```
IF logical-expression
    instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

类似这样的嵌套结构最多可以嵌套到 256 层深。

使用 ELIF 可以更简单地编写相同的结构:

```
IF logical-expression
    instructions
ELIF logical-expression2
    instructions
ELIF logical-expression3
    instructions
ENDIF
```

对于 IF...ENDIF 指令对，此结构只在当前嵌套深度上增加一层。

示例

示例 7-3 中，如果定义了 NEWVERSION，则汇编第一组指令，否则汇编另一组指令。

示例 7-3 根据变量是否定义执行条件汇编

```
        IF :DEF:NEWVERSION
            ; first set of instructions/directives
        ELSE
            ; alternative set of instructions/directives
        ENDIF
```

按如下方式调用 armasm 来定义 NEWVERSION，将汇编第一组指令和命令：

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

按如下方式调用 armasm 时未定义 NEWVERSION，则汇编第二组指令和命令：

```
armasm test.s
```

示例 7-4 中，如果 NEWVERSION 的值为 {TRUE}，则汇编第一组指令，否则汇编另一组指令。

示例 7-4 根据变量的值执行条件汇编

```
        IF NEWVERSION = {TRUE}
            ; first set of instructions/directives
        ELSE
            ; alternative set of instructions/directives
        ENDIF
```

按如下方式调用 armasm，将汇编第一组指令和命令：

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

按如下方式调用 armasm，将汇编第二组指令和命令：

```
armasm --predefine "NEWVERSION SETL {FALSE}" test.s
```

7.4.5 WHILE 和 WEND

`WHILE` 指令开始一个要重复汇编的指令或命令序列。该序列用 `WEND` 指令终止。

语法

`WHILE` *logical-expression*

code

`WEND`

其中:

logical-expression

是一个取值为 `{TRUE}` 或 `{FALSE}` 的表达式(请参阅第3-32 页的*逻辑表达式*)。

用法

将 `WHILE` 指令与 `WEND` 指令一起使用,可以多次汇编一个指令序列。重复次数可以是零。

可以在 `WHILE...WEND` 循环内使用 `IF...ENDIF` 条件。

可以嵌套 `WHILE...WEND` 循环(请参阅第7-31 页的*嵌套指令*)。

示例

```
count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case,
            ; code                  ; this code will be
            ; code                  ; repeated four times
        WEND
```


7.5 Frame 指令

本节介绍下列指令:

- 第7-42 页的*FRAME ADDRESS*
- 第7-43 页的*FRAME POP*
- 第7-44 页的*FRAME PUSH*
- 第7-46 页的*FRAME REGISTER*
- 第7-47 页的*FRAME RESTORE*
- 第7-48 页的*FRAME RETURN ADDRESS*
- 第7-49 页的*FRAME SAVE*
- 第7-50 页的*FRAME STATE REMEMBER*
- 第7-51 页的*FRAME STATE RESTORE*
- 第7-52 页的*FRAME UNWIND ON*
- 第7-52 页的*FRAME UNWIND OFF*
- 第7-53 页的*FUNCTION 或 PROC*
- 第7-54 页的*ENDFUNC 或 ENDP*。

正确使用下列指令可以:

- 将 `armlink --callgraph` 选项用于计算汇编程序函数的堆栈使用量。

  以下规则用于确定堆栈使用量:

  — 如果函数未标有 `PROC` 或 `ENDP`，则堆栈的使用量未知。

  — 如果函数标有 `PROC` 或 `ENDP`，但没有 `FRAME PUSH` 或 `FRAME POP`，则堆栈使用量假定为零。这意味着无需手动添加 `FRAME PUSH 0` 或 `FRAME POP 0`。

  — 如果函数标有 `PROC` 或 `ENDP`，并且有 `FRAME PUSH n` 或 `FRAME POP n`，则堆栈使用量假定为 `n` 个字节。

- 帮助避免函数构造中的错误，特别是在修改现有代码时
- 使汇编程序对函数构造中的错误发出警告
- 在调试时启用函数调用的回溯跟踪
- 启用调试器剖析汇编程序函数。

如果需要剖析汇编程序函数，但出于其他目的而不需要帧描述指令:

- 必须使用 FUNCTION 和 ENDFUNC 指令，或者使用 PROC 和 ENDP 指令
- 可以忽略其他 FRAME 指令
- 您只需为要剖析的函数使用 FUNCTION 和 ENDFUNC 指令。

在 DWARF 中，规范帧地址是堆栈上的一个地址，它指定一个被中断函数的调用帧的位置。

7.5.1 FRAME ADDRESS

FRAME ADDRESS 指令说明如何为后面的指令计算规范帧地址。只能在含有 FUNCTION 和 ENDFUNC 指令，或含有 PROC 和 ENDP 指令的函数内使用它。

语法

FRAME ADDRESS *reg*[,*offset*]

其中：

reg 是规范帧地址所基于的寄存器。这一般是 sp，除非函数使用了其他帧指针。

offset 是规范帧地址相对 *reg* 的偏移量。如果 *offset* 为 0，则可以省略它。

用法

如果代码改变了规范帧地址所基于的寄存器，或者如果代码改变了规范帧地址相对该寄存器的偏移量，则可使用 FRAME ADDRESS。必须在改变规范帧地址计算方式的指令后面，立即使用 FRAME ADDRESS。

注意

如果代码使用单个指令来保存寄存器及改变堆栈指针，则可以使用 FRAME PUSH 来代替 FRAME ADDRESS 和 FRAME SAVE（请参阅第7-44 页的*FRAME PUSH*）。

如果代码使用单个指令来加载寄存器及改变堆栈指针，则可以使用 FRAME POP 来代替 FRAME ADDRESS 和 FRAME RESTORE（请参阅第7-43 页的*FRAME POP*）。

示例

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of sp on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of sp
```

7.5.2 FRAME POP

当被调用方重新加载寄存器时，可使用 `FRAME POP` 指令来通知汇编程序。只能在含有 `FUNCTION` 和 `ENDFUNC` 指令，或者 `PROC` 和 `ENDP` 指令的函数内使用它。

在函数的最后一条指令后不需要这样做。

语法

`FRAME POP` 有三种备选语法:

`FRAME POP {`*`reglist`*`}`

`FRAME POP {`*`reglist`*`},`*`n`*

`FRAME POP` *`n`*

其中:

reglist	是恢复为进入函数时其值的寄存器列表。列表中必须至少有一个寄存器。
n	是堆栈指针移动的字节数。

用法

`FRAME POP` 等效于 `FRAME ADDRESS` 和 `FRAME RESTORE` 指令。当单个指令加载寄存器及改变堆栈指针时，可以使用它。

必须在其涉及到的指令后立即使用 `FRAME POP`。

如果 *n* 未指定或为零，则汇编程序将计算规范帧地址相对于 `{`*`reglist`*`}` 的新偏移量。它假定:

- 弹出的每个 ARM 寄存器占用堆栈上的四个字节
- 弹出的每个 VFP 单精度寄存器占用堆栈上的四个字节，每个列表再加一个四字节字
- 弹出的每个 VFP 双精度寄存器占用堆栈上的八个字节，每个列表再加一个四字节字。

请参阅第7-42 页的*FRAME ADDRESS* 和第7-47 页的*FRAME RESTORE*。

7.5.3 FRAME PUSH

当被调用方保存寄存器时（通常在进入函数时），可使用 FRAME PUSH 指令来通知汇编程序。只能在含有 FUNCTION 和 ENDFUNC 指令，或者 PROC 和 ENDP 指令的函数内使用它。

语法

FRAME PUSH 有两种备选语法:

```
FRAME PUSH {reglist}
```

```
FRAME PUSH {reglist},n
```

```
FRAME PUSH n
```

其中:

reglist 是在规范帧地址后连续存储的寄存器列表。列表中必须至少有一个寄存器。

n 是堆栈指针移动的字节数。

用法

FRAME PUSH 等效于 FRAME ADDRESS 和 FRAME SAVE 指令。当单个指令保存寄存器及改变堆栈指针时，可以使用它。

必须在其涉及到的指令后立即使用 FRAME PUSH。

如果 *n* 未指定或为零，则汇编程序将计算规范帧地址相对于 {*reglist*} 的新偏移量。它假定:

- 存入堆栈的每个 ARM 寄存器占用堆栈上的四个字节
- 存入堆栈的每个 VFP 单精度寄存器占用堆栈上的四个字节，每个列表再加一个四字节字
- 弹出的每个 VFP 双精度寄存器占用堆栈上的八个字节，每个列表再加一个四字节字。

请参阅第7-42 页的*FRAME ADDRESS* 和第7-49 页的*FRAME SAVE*。

示例

```
p   PROC ; Canonical frame address is sp + 0
    EXPORT  p
    PUSH    {r4-r6,lr}
         ; sp has moved relative to the canonical frame address,
         ; and registers r4, r5, r6 and lr are now on the stack
    FRAME PUSH {r4-r6,lr}
         ; Equivalent to:
         ; FRAME ADDRESS    sp,16       ; 16 bytes in {r4-r6,lr}
         ; FRAME SAVE    {r4-r6,lr},-16
```


7.5.4 FRAME REGISTER

使用 `FRAME REGISTER` 指令可保存寄存器中存放的函数自变量的位置记录。只能在含有 `FUNCTION` 和 `ENDFUNC` 指令，或者 `PROC` 和 `ENDP` 指令的函数内使用它。

语法

`FRAME REGISTER` *reg*1,*reg2*

其中:

reg1 是存放函数入口自变量的寄存器。

reg2 是用来保存参数值的寄存器。

用法

当使用寄存器保存在进入函数时存放在另一个寄存器中的自变量时，可使用 `FRAME REGISTER` 指令。

7.5.5 FRAME RESTORE

使用 `FRAME RESTORE` 指令可通知汇编程序，指定寄存器的内容已被恢复为进入函数时保存的值。只能在含有 `FUNCTION` 和 `ENDFUNC` 指令，或者 `PROC` 和 `ENDP` 指令的函数内使用它。

语法

```
FRAME RESTORE {reglist}
```

其中:

reglist 是已恢复其内容的寄存器列表。列表中必须至少有一个寄存器。

用法

在被调用方从堆栈中重新加载寄存器后，立即使用 `FRAME RESTORE`。在函数的最后一条指令后不需要这样做。

reglist 可以包含整数寄存器或浮点寄存器，但不能同时包含两者。

注意

如果代码使用单个指令来加载寄存器及改变堆栈指针，则可以使用 `FRAME POP` 来代替 `FRAME RESTORE` 和 `FRAME ADDRESS`（请参阅第7-43 页的*FRAME POP*）。

7.5.6 FRAME RETURN ADDRESS

`FRAME RETURN ADDRESS` 指令为使用除 r14 之外的寄存器的函数提供其返回地址。只能在含有 `FUNCTION` 和 `ENDFUNC` 指令，或者 `PROC` 和 `ENDP` 指令的函数内使用它。

——注意——

将除 r14 之外的寄存器用于函数返回地址的任何函数均不符合 AAPCS。这种函数不能被导出。

语法

`FRAME RETURN ADDRESS` *`reg`*

其中:

reg 是用于返回地址的寄存器。

用法

在未将 r14 用于函数返回地址的任何函数中，都可使用 `FRAME RETURN ADDRESS` 指令。否则，调试器不能回溯跟踪该函数。

在引入该函数的 `FUNCTION` 或 `PROC` 指令后立即使用 `FRAME RETURN ADDRESS`。

7.5.7 FRAME SAVE

`FRAME SAVE` 指令描述保存的寄存器内容相对于规范帧地址的位置。只能在含有 `FUNCTION` 和 `ENDFUNC` 指令，或者 `PROC` 和 `ENDP` 指令的函数内使用它。

语法

```
FRAME SAVE {reglist}, offset
```

其中:

reglist 是从规范帧地址偏移 *offset* 处开始连续存储的寄存器列表。列表中必须至少有一个寄存器。

用法

在被调用方将寄存器存储到堆栈中后，立即使用 `FRAME SAVE`。

reglist 可以包含不需要回溯跟踪的寄存器。汇编程序确定需要在 DWARF 调用帧信息中记录哪些寄存器。

——注意——

如果代码使用单个指令来保存寄存器及改变栈指针，则可以使用 `FRAME PUSH` 来代替 `FRAME SAVE` 和 `FRAME ADDRESS`（请参阅第7-44 页的*FRAME PUSH*）。

7.5.8 FRAME STATE REMEMBER

FRAME STATE REMEMBER 指令保存有关如何计算规范帧地址以及已保存寄存器值的位置的当前信息。只能在含有 FUNCTION 和 ENDFUNC 指令，或者 PROC 和 ENDP 指令的函数内使用它。

语法

FRAME STATE REMEMBER

用法

在执行内联退出序列的过程中，会改变有关规范帧地址的计算信息以及已保存的寄存器值的位置信息。在退出序列之后，另一个跳转可以继续像以前一样使用相同的信息。使用 FRAME STATE REMEMBER 可保存这些信息，使用 FRAME STATE RESTORE 来进行恢复。

这些指令可以嵌套。每个 FRAME STATE RESTORE 指令必须有对应的 FRAME STATE REMEMBER 指令。请参阅:

- 第7-51 页的*FRAME STATE RESTORE*
- 第7-53 页的*FUNCTION 或 PROC*。

示例

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; do not have to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        POP     {r4-r6,pc}
        ENDP
```

7.5.9 FRAME STATE RESTORE

`FRAME STATE RESTORE` 指令恢复有关如何计算规范帧地址和已保存寄存器值的位置的信息。只能在含有 `FUNCTION` 和 `ENDFUNC` 指令，或者 `PROC` 和 `ENDP` 指令的函数内使用它。

语法

```
FRAME STATE RESTORE
```

用法

请参阅:

- 第7-50 页的*FRAME STATE REMEMBER*
- 第7-53 页的*FUNCTION 或 PROC*。

7.5.10 FRAME UNWIND ON

`FRAME UNWIND ON` 指令指示汇编程序为此函数以及后续函数生成*展开*表。

语法

```
FRAME UNWIND ON
```

用法

您可以在函数外使用此指令。在这种情况下，汇编程序为所有后续函数生成*展开*表，直到到达 `FRAME UNWIND OFF` 指令。

另请参阅第3-16 页的*控制异常表生成*。

7.5.11 FRAME UNWIND OFF

`FRAME UNWIND OFF` 指令指示汇编程序为此函数以及后续函数生成*非展开*表。

语法

```
FRAME UNWIND OFF
```

用法

您可以在函数外使用此指令。在这种情况下，汇编程序为所有后续函数生成*非展开* 73表，直到到达 `FRAME UNWIND ON` 指令。

另请参阅第3-16 页的*控制异常表生成*。

7.5.12 FUNCTION 或 PROC

`FUNCTION` 指令标记函数的开始。`PROC` 是 `FUNCTION` 的同义词。

语法

```
label FUNCTION [{reglist1} [, {reglist2}]]
```

其中:

reglist1 是由被调用方保存的 ARM 寄存器的可选列表。如果 *reglist1* 不存在，并且调试器检查寄存器使用情况，则调试器假定正在使用 AAPCS。

reglist2 是由被调用方保存的 VFP 寄存器的可选列表。

用法

可使用 `FUNCTION` 标记函数的开始。在为 ELF 生成 DWARF 调用帧信息时，汇编程序使用 `FUNCTION` 来标识一个函数的开始。

`FUNCTION` 将规范帧地址设置为 r13 (sp)，并将帧状态堆栈清空。

每个 `FUNCTION` 指令必须有一个匹配的 `ENDFUNC` 指令。不能嵌套 `FUNCTION/ENDFUNC` 对，并且它们不能包含 `PROC` 或 `ENDP` 指令。

如果要使用自己的过程调用标准，可以使用可选的 *reglist* 参数通知调试器此标准。并非所有调试器都支持此功能。有关详细信息，请参阅所用调试器的说明文档。

另请参阅第7-42 页的*FRAME ADDRESS* 到第7-51 页的*FRAME STATE RESTORE*。

——注意——

`FUNCTION` 不会自动对齐字边界（或 Thumb 的半字边界）。如有必要，可使用 `ALIGN` 确保对齐，否则调用帧将无法指向函数的开始处。有关详细信息，请参阅第7-63 页的*ALIGN*。

示例

```
        ALIGN      ; ensures alignment
dadd    FUNCTION   ; without the ALIGN directive, this might not be
word-aligned
        EXPORT  dadd
        PUSH       {r4-r6,lr}    ; this line automatically word-aligned
        FRAME PUSH {r4-r6,lr}
        ; subroutine body
        POP        {r4-r6,pc}
        ENDFUNC

func6   PROC {r4-r8,r12},{D1-D3} ; non-AAPCS-conforming function
        ...
        ENDP
```

7.5.13 ENDFUNC 或 ENDP

`ENDFUNC` 指令标记符合 AAPCS 的函数的结尾（请参阅第7-53 页的*FUNCTION 或 PROC*）。`ENDP` 是 `ENDFUNC` 的同义词。

7.6 报告指令

本节介绍下列指令:

- *ASSERT*
  在汇编时，如果断言为 false，则生成错误消息。
- 第7-56 页的*INFO*
  在汇编时产生诊断消息。
- 第7-57 页的*OPT*
  设置列表选项。
- 第7-59 页的*TTL 和 SUBT*
  在列表中插入标题和小标题。

7.6.1 ASSERT

在第二轮汇编时，如果给定的断言为 false，则 `ASSERT` 指令会生成错误消息。

语法

`ASSERT` *`logical-expression`*

其中:

`logical-expression`
是一个取值为 `{TRUE}` 或 `{FALSE}` 的断言。

用法

使用 `ASSERT` 可确保在汇编时，任何必要的条件能得以满足。

如果断言为 false，则会生成错误消息，并且汇编失败。

另请参阅第7-56 页的*INFO*。

示例

```
        ASSERT  label1 <= label2    ; Tests if the address
                                    ; represented by label1
                                    ; is <= the address
                                    ; represented by label2.
```

7.6.2 INFO

`INFO` 指令支持在每一轮汇编过程中产生诊断消息。

`!` 与 `INFO` 非常相似，但输出报告相对简单。

语法

`INFO` *`numeric-expression, string-expression`*

其中：

`numeric-expression`

是在汇编时求值的一个数值表达式。如果该表达式的值等于零，则：

- 在第一轮汇编时不执行操作
- 在第二轮汇编时显示 *`string-expression`*。

如果该表达式的值不等于零，则显示 *`string-expression`* 作为错误消息，并且汇编失败。

`string-expression`

是取值为字符串的表达式。

用法

`INFO` 提供了灵活的创建自定义错误消息的方式。有关数值表达式和字符串表达式的其他信息，请参阅第3-29 页的*数字表达式*和第3-28 页的*字符串表达式*。

另请参阅第7-55 页的*ASSERT*。

示例

```
INFO    0, "Version 1.0"

IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```


7.6.3 OPT

`OPT` 指令从源代码内设置列表选项。

语法

`OPT n`

其中:

n 是 `OPT` 指令设置。表7-2 列出了有效的设置。

表7-2 OPT 指令设置

OPT n	效果
1	启用常规列表。
2	关闭常规列表。
4	进页。产生一个立即换页并开始一个新页。
8	将行号计数器重置为零。
16	启用 `SET`、`GBL` 和 `LCL` 指令的列表。
32	关闭 `SET`、`GBL` 和 `LCL` 指令的列表。
64	启用宏扩展的列表。
128	关闭宏扩展的列表。
256	启用宏调用的列表。
512	关闭宏调用的列表。
1024	启用第一轮汇编列表。
2048	关闭第一轮汇编列表。
4096	启用条件指令的列表。
8192	关闭条件指令的列表。
16384	启用 `MEND` 指令的列表。
32768	关闭 `MEND` 指令的列表。

用法

指定 `--list=` 汇编程序选项可启用列表。

缺省情况下，`--list=` 选项产生一个常规列表，其中包含变量声明、宏扩展、调用条件指令以及 `MEND` 指令。列表仅在第二轮汇编时产生。使用 `OPT` 指令在代码内修改缺省列表选项。有关 `--list=` 选项的信息，请参阅第3-12 页的*列表输出到文件中*。

可以使用 `OPT` 来格式化代码列表。例如，可以在函数和代码节前指定一个新页。

示例

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4               ; places a page break before func1
func1   ; code
```

7.6.4 TTL 和 SUBT

TTL 指令在列表文件的每页开头插入一个标题。每页都显示该标题，直到发出新的 TTL 指令。

SUBT 指令在列表文件的各页上放置一个小标题。每页都显示该小标题，直到发出新的 SUBT 指令。

语法

TTL *title*

SUBT *subtitle*

其中:

title 是标题。

subtitle 是小标题。

用法

使用 TTL 指令可在列表文件的各页顶部放置一个标题。如果想让标题显示在第一页上，则 TTL 指令必须位于源文件的第一行。

使用附加的 TTL 指令可更改标题。每个新的 TTL 指令从下一页的顶部开始生效。

使用 SUBT 指令可在列表文件的各页顶部放置一个小标题。小标题显示在标题下面的行中。如果想让小标题显示在第一页上，则 SUBT 指令必须位于源文件的第一行。

使用附加的 SUBT 指令可更改小标题。每个新的 SUBT 指令从下一页的顶部开始生效。

示例

```
        TTL     First Title     ; places a title on the first
                                ; and subsequent pages of a
                                ; listing file.
        SUBT    First Subtitle  ; places a subtitle on the
                                ; second and subsequent pages
                                ; of a listing file.
```

7.7 指令集和语法选择指令

本节介绍下列指令:

- 第7-61 页的*ARM*、*THUMB*、*THUMBX*、*CODE16* 和 *CODE32*。

7.7.1 ARM、THUMB、THUMBX、CODE16 和 CODE32

ARM 指令和 CODE32 指令是同义词。它们指示汇编程序将后面的指令解释为 32 位 ARM 指令。必要时，它们也可插入最多三个填充字节，以对齐到下一个字边界。

在此模式下，汇编程序接受最新版本和旧版本的汇编语言。

语法

```
ARM
THUMB
THUMBX
CODE16
CODE32
```

用法

在包含使用不同指令集的代码的文件中：

- ARM 必须位于任何 ARM 代码之前。CODE32 是 ARM 的同义词。
- THUMB 必须位于用新语法编写的 Thumb 代码之前。
- THUMBX 必须位于用新语法编写的 Thumb-2EE 代码之前。
- CODE16 必须位于用旧 Thumb 语法编写的 Thumb 代码之前。

这些命令不汇编为改变状态的指令。它们仅指示汇编程序适当地汇编 ARM、Thumb-2、Thumb-2EE 或 Thumb 指令，并在必要时插入填充字节。

示例

此示例演示如何使用 ARM 和 CODE16 从 ARM 指令跳转到 16 位 Thumb 指令。

```
        AREA ToThumb, CODE, READONLY    ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are ARM
start
        ADR     r0, into_thumb + 1      ; Processor starts in ARM state
        BX      r0                      ; Inline switch to Thumb state

        THUMB                           ; Subsequent instructions are Thumb
into_thumb
        MOVS    r0, #10                 ; New-style Thumb instructions
```

7.8 其他指令

本节介绍下列指令:

7.8.1 ALIGN

`ALIGN` 指令通过用零或 `NOP` 指令进行填充来使当前位置与指定的边界对齐。

语法

```
ALIGN {expr{,offset{,pad {, padsize }}}}
```

其中:

expr 是一个数值表达式，取值为 2^0 到 2^{31} 范围内的 2 的任何次幂

offset 可以是任何数值表达式

pad 可以是任何数值表达式

padsize 可为 1、2 或 4。

操作

使当前位置对齐到如下形式的下一地址:

```
offset + n * expr
```

如果未指定 *expr*，则 `ALIGN` 会将当前位置设置到下一个字（四字节）边界处。前一个位置和当前新位置之间的未用空间用以下内容填充:

- 如果指定了 *pad*，则用 *pad* 的副本填充满足以下所有条件时，用
- `NOP` 指令填充:
  - — 未指定 *pad*
  - — ARM 或 Thumb 指令后面是 `ALIGN` 指令
  - — 在当前节中，`AREA` 指令设置了 `CODEALIGN` 属性
- 其他情况用零填充。

根据 *padsize* 值的情况，*pad* 将被分别视为一个字节、半字或字。如果未指定 *padsize*，则 *pad* 在数据节中缺省为字节，在 Thumb 代码中缺省为半字，在 ARM 代码中缺省为字。

用法

使用 `ALIGN` 可确保数据和代码对齐到适当的边界上。在下列情况下，这通常是必须的:

- `ADR` Thumb 伪指令只能加载字对齐的地址，但 Thumb 代码内的标签可能不是字对齐的。使用 `ALIGN 4` 可确保 Thumb 代码内的地址是四字节对齐的。

- 使用 ALIGN 可利用有些 ARM 处理器上的高速缓存。例如，ARM940T 带有一个含 16 字节行的高速缓存。使用 ALIGN 16 可在 16 字节边界上对齐函数入口点，并使高速缓存的效率最高。
- LDRD 和 STRD 双字数据传送必须是八字节对齐的。如果要用 LDRD 或 STRD 访问数据，则在内存分配指令（如 DCQ）之前使用 ALIGN 8（请参阅第7-17 页的*数据定义指令*）。
- 只有标签的行可以是任意对齐的。随后的 ARM 代码是字对齐的（Thumb 代码是半字对齐的）。因此标签不能正确寻址代码。在标签前使用 ALIGN 4（或对 Thumb 代码使用 ALIGN 2）。

对齐相对于例程所在的 ELF 节的起始位置。节必须对齐到相同的或更近似的边界上。AREA 指令中的 ALIGN 属性以不同方式指定（请参阅第7-65 页的*AREA* 和*示例*）。

示例

```
        AREA    cacheable, CODE, ALIGN=3
rout1   ; code              ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr       ; aligned only on 4-byte boundary
        ALIGN   8           ; now aligned on 8-byte boundary
rout2   ; code
```

```
        AREA    OffsetExample, CODE
        DCB     1           ; This example places the two
        ALIGN   4,3         ; bytes in the first and fourth
        DCB     1           ; bytes of the same word.
```

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1           ; pc now misaligned
        ALIGN               ; ensures that subroutine1 addresses
subroutine1                 ; the following instruction.
        MOV r5,#0x5
```


7.8.2 AREA

AREA 指令指示汇编程序汇编新的代码节或数据节。节是不可分的已命名独立代码或数据块，它们由链接器处理。有关详细信息，请参阅第2-14 页的*ELF 节和 AREA 指令*。

语法

```
AREA sectionname{,attr}{,attr}...
```

其中:

sectionname 是将要指定的节名。

可以为节选择任何名称。但是，以数字开始的名称必须包含在竖杠内，否则会产生一个缺失节名错误。例如，`|1_DataArea|`。

有些名称是习惯性的名称。例如，`|.text|` 用于表示由 C 编译器生成的代码节，或以某种方式与 C 库关联的代码节。

attr 是一个或多个用逗号分隔的节属性。有效的属性有:

`ALIGN=`*expression*

缺省情况下，ELF 节在四字节边界上对齐。*expression* 可以取值 0 到 31 之间的任何整数。节在 $2^{expression}$ 字节边界上对齐。例如，如果 *expression* 是 10，则节在 1KB 边界上对齐。

这与 `ALIGN` *指令所指定的方式不同*。请参阅第7-63 页的*ALIGN*。

——注意——

不要对 ARM 代码节使用 `ALIGN=0` 或 `ALIGN=1`。

不要对 Thumb 代码节使用 `ALIGN=0`。

`ASSOC=`*section*

section 指定一个关联的 ELF 节。*sectionname* 必须包含在含有 *section* 的任何链接中

`CODE` 包含机器指令。`READONLY` 是缺省值。

`CODEALIGN`

当在节内的 ARM 或 Thumb 指令后使用 `ALIGN` 指令时，该属性导致汇编程序插入 `NOP` 指令，除非 `ALIGN` 指令指定了其他填充方式。

`COMDEF` 是一个公共节定义。此 ELF 节可以包含代码或数据。它必须等同于其他源文件中拥有相同名称的任何其他节。

名称相同的同一 ELF 节在内存的同一节中被链接器覆盖。如果有任何不同，则链接器会产生一个警告，并且不覆盖这些节。请参阅 *《RealView 编译工具链接器和实用程序指南》*中的第 3 章 *使用基本链接器功能*。

`COMGROUP=`*`symbol_name`*

是一个公共组节。公共组中的所有节都是公共的。当对象被链接后，其他目标文件可能具有带有 *`symbol_name`* 签名的一个 GROUP。最终映像中只包含一个组。

`COMMON` 是一个公共数据节。不能在其中定义任何代码或数据。它由链接器初始化为零。名称相同的所有公共节在内存的同一节中被链接器覆盖。它们并不都必须具有相同大小。链接器按每个名称的最大公共节的需要分配空间。

`DATA` 包含数据，不包含指令。`READWRITE` 是缺省值。

`GROUP=`*`symbol_name`*

是组的签名，它必须由源文件或源文件中包含的文件定义。具有相同 *`symbol_name`* 签名的所有 `AREAS` 都被置于同一组中。组内的各节同时保存或显现。

`NOALLOC` 指示在目标系统上没有为此区域分配内存。

`NOINIT` 指示数据节未初始化，或初始化为零。它只包含空间保留指令 `SPACE` 或初始化值为零的 `DCB`、`DCD`、`DCDU`、`DCQ`、`DCQU`、`DCW` 或 `DCWU`。您可以在链接时决定某区域是未初始化还是初始化为零（请参阅 *《RealView 编译工具链接器和实用程序指南》*中的第 3 章 *使用基本链接器功能*）。

`READONLY` 指示不应向此节写入。这是代码区域的缺省值。

`READWRITE` 指示可以读写此节。这是数据区域的缺省值。

用法

使用 `AREA` 指令可将源文件细分为 ELF 节。可以在多个 `AREA` 指令中使用相同的名称。名称相同的所有区域都放在相同的 ELF 节中。只有特定名称的第一个 `AREA` 指令的属性才会被应用。

通常应对代码和数据使用不同的 ELF 节。大型程序通常可方便地划分为多个代码节。大量独立的数据集通常也最好放在不同的节中。

局部标签的作用域是由 `AREA` 指令定义的，并可选择用 `ROUT` 指令细分（请参阅第3-26 页的*局部标签*和第7-80 页的*ROUT*）。

一组汇编代码必须至少有一个 `AREA` 指令。

示例

下列示例定义名为 `Example` 的只读代码节。

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```


7.8.3 END

END 指令通知汇编程序它已到达源文件的末尾。

语法

```
END
```

用法

每个汇编语言源文件都必须以一行单独的 END 结束。

如果源文件已被 GET 指令包含在父文件中，则汇编程序会返回到父文件，并在 GET 指令后的第一行继续汇编。有关详细信息，请参阅第7-73 页的*GET 或 INCLUDE*。

如果在第一轮汇编时到达顶层源文件的 END 指令而没有出现任何错误，则开始第二轮汇编。

如果在第二轮汇编时到达顶层源文件的 END 指令，则汇编程序完成汇编，并写入适当的输出。

7.8.4 ENTRY

ENTRY 指令声明程序的入口点。

语法

```
ENTRY
```

用法

必须为一个程序指定至少一个 ENTRY 点。如果不存在 ENTRY，则链接时会产生一个警告。

在一个源文件内不能使用多个 ENTRY 指令。并非每个源文件都必须包含 ENTRY 指令。如果在一个源文件内有多个 ENTRY 指令，则汇编时会产生错误消息。

示例

```
        AREA    ARMex, CODE, READONLY
        ENTRY                 ; Entry point for the application
```


7.8.5 EQU

EQU 指令为数值常数、寄存器相对的值或程序相对的值指定一个符号名称。* 是 EQU 的同义词。

语法

name EQU *expr*{, *type*}

其中:

name 是要为值指定的符号名称。

expr 是一个寄存器相对的地址、程序相对的地址、绝对地址或 32 位整型常数。

type 是可选的。*type* 可为下列值之一:

- ARM
- THUMB
- CODE32
- CODE16
- DATA

仅当 *expr* 是一个绝对地址时,才能使用 *type*。如果导出了 *name*,则会根据 *type* 的值,将目标文件的符号表中的 *name* 条目标记为 ARM、THUMB、CODE32、CODE16 或 DATA。这些信息可由链接器使用。

用法

使用 EQU 可定义常数。这类似于在 C 中使用 **#define** 定义常数。

有关导出符号的信息,请参阅第7-77 页的*KEEP* 和第7-70 页的*EXPORT 或 GLOBAL*。

示例

```
abc EQU 2               ; assigns the value 2 to the symbol abc.

xyz EQU label+8         ; assigns the address (label+8) to the
                        ; symbol xyz.

fiq EQU 0x1C, CODE32    ; assigns the absolute address 0x1C to
                        ; the symbol fiq, and marks it as code
```


7.8.6 EXPORT 或 GLOBAL

EXPORT 指令声明一个符号，可由链接器用于解析不同的对象和库文件中的符号引用。GLOBAL 是 EXPORT 的同义词。

语法

```
EXPORT {[WEAK]}
EXPORT symbol {[attr]}
EXPORT symbol [WEAK{,attr}]
```

其中:

symbol 是要导出的符号名称。符号名区分大小写。如果省略了 *symbol*，则导出所有符号。

WEAK 仅当没有其他源导出另一个 *symbol* 时，才应将此 *symbol* 导入其他源中。如果使用了不带 *symbol* 的 [WEAK]，则所有导出的符号都是处于次要地位的。

attr 是下列项之一:

- DYNAMIC 当源代码链接到动态组件中时，*symbol* 对于其他组件是可见的。
- PROTECTED 当源代码链接到动态组件中时，*symbol* 对于其他组件是可见的，但是不能由其他组件重新定义。
- HIDDEN 当源代码链接到动态组件中时，*symbol* 对于其他组件是不可见的。

用法

使用 `EXPORT` 可使其他文件中的代码能访问当前文件中的符号。

使用 `[WEAK]` 属性可通知链接器，如果可以使用其他源中的不同 *`symbol`* 实例，则不同实例将优先于此实例。`[WEAK]` 属性可与任何符号可见性属性一起使用。

另请参阅第7-74 页的*IMPORT 和 EXTERN*。

示例

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     r0,r0,r1
```

重复导出可覆盖符号可见性。在以下示例中，最后一个 `EXPORT` 在绑定和可见性上优先:

```
        EXPORT  SymA[WEAK]      ; Export as weak-hidden
        EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```


7.8.7 EXPORTAS

EXPORTAS 指令允许将符号导出到目标文件中，该符号与对应的源文件中的符号不同。

语法

EXPORTAS *symbol1*, *symbol2*

其中:

symbol1 是源文件中的符号名称。*symbol1* 必须已定义。它可以是任何符号，包括区域名、标签或常数。

symbol2 是希望在目标文件中出现的符号名称。

符号名区分大小写。

用法

使用 EXPORTAS 可改变目标文件中的符号，而不必改变源文件中的每个实例。

另请参阅第7-70 页的*EXPORT 或 GLOBAL*。

示例

```
    AREA data1, DATA       ;; starts a new area data1
    AREA data2, DATA       ;; starts a new area data2
    EXPORTAS data2, data1  ;; the section symbol referred to as data2 will
                           ;; appear in the object file string table as data1.

one EQU  2
    EXPORTAS one, two
    EXPORT one             ;; the symbol 'two' will appear in the object
                           ;; file's symbol table with the value 2.
```


7.8.8 GET 或 INCLUDE

GET 指令在被汇编的文件内包含一个文件。所包含的文件在 GET 指令的位置处汇编。INCLUDE 是 GET 的同义词。

语法

GET *filename*

其中:

filename 是要在汇编中包含的文件名称。汇编程序接受 UNIX 或 MS-DOS 格式的路径名。

用法

GET 对在汇编代码中包含宏定义、EQU 指令和存储器映射很有用。当完成所包含文件的汇编后，在 GET 指令后的下一行继续汇编。

缺省情况下，汇编程序在当前位置搜索所包含的文件。当前位置即调用文件所在的目录。使用 -i 汇编程序命令行选项可向搜索路径添加目录。包含空格的文件名和目录名不能括在双引号 (" ") 内。

所包含的文件可包含其他 GET 指令以包含其他文件（请参阅第7-31 页的*嵌套指令*）。

如果所包含的文件位于与当前位置不同的目录中，则该目录就成为当前位置，直到所包含的文件结束。原先的当前位置随后恢复。

GET 不能用于包含目标文件（请参阅第7-76 页的*INCBIN*）。

示例

```
    AREA    Example, CODE, READONLY
    GET     file1.s               ; includes file1 if it exists
                                  ; in the current place.
    GET     c:\project\file2.s    ; includes file2
    GET     c:\Program files\file3.s  ; space is permitted
```

7.8.9 IMPORT 和 EXTERN

这些指令为汇编程序提供一个未在当前汇编中定义的名称。

IMPORT 将导入名称，不管该名称在当前汇编中是否被引用。

EXTERN 仅导入在当前汇编中引用的名称。

语法

IMPORT *symbol* {[*attr*]}

IMPORT *symbol* [WEAK{,*attr*}]

EXTERN *symbol* {[*attr*]}

EXTERN *symbol* [WEAK{,*attr*}]

其中:

symbol 是在单独汇编的源文件、目标文件或库中定义的一个符号名称。符号名区分大小写。

WEAK 防止链接器在符号未在其他地方定义时产生错误消息。同时防止链接器搜索还未包含的库。

attr 是下列项之一:

- DYNAMIC 当源代码链接到动态组件中时，*symbol* 对于其他组件是可见的。
- PROTECTED 当源代码链接到动态组件中时，*symbol* 对于其他组件是可见的，但是不能由其他组件重新定义。
- HIDDEN 当源代码链接到动态组件中时，*symbol* 对于其他组件是不可见的。

用法

在链接时，名称被解析为在其他目标文件中定义的符号。该符号被当作程序地址。如果未指定 [WEAK] 且在链接时没有找到相应的符号，则链接器会产生错误。

如果指定了 [WEAK] 且在链接时没有找到相应的符号：

- 如果该引用是 B 或 BL 指令的目标，则将下一指令的地址作为该符号的值。这样就有效地使 B 或 BL 指令变成一个 NOP。
- 否则，该符号的值取为零。

示例

此示例测试是否已链接 C++ 库，并根据结果执行条件跳转。

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the address of
                                ; __CPP_INITIALIZE function.
LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
CMP     r0,#0                   ; Test if zero.
BEQ     nocplusplus             ; Branch on the result.
```


7.8.10 INCBIN

`INCBIN` 指令在被汇编的文件内包含一个文件。该文件按原样包含，没有进行汇编。

语法

`INCBIN` *`filename`*

其中:

`filename` 是要在汇编中包含的文件名称。汇编程序接受 UNIX 或 MS-DOS 格式的路径名。

用法

可以使用 `INCBIN` 来包含可执行文件、文字或其他任意数据。文件的内容将按字节逐一添加到当前 ELF 节中，而不进行任何方式的解释。汇编在 `INCBIN` 指令的下一行继续执行。

缺省情况下，汇编程序在当前位置搜索所包含的文件。当前位置即调用文件所在的目录。使用 `-i` 汇编程序命令行选项可向搜索路径添加目录。包含空格的文件名和目录名不能括在双引号 (" ") 内。

示例

```
        AREA    Example, CODE, READONLY
        INCBIN  file1.dat               ; includes file1 if it
                                        ; exists in the
                                        ; current place.
        INCBIN  c:\project\file2.txt    ; includes file2
```


7.8.11 KEEP

KEEP 指令指示汇编程序在目标文件的符号表中保留局部符号。

语法

KEEP {*symbol*}

其中:

symbol 是要保留的局部符号的名称。如果未指定 *symbol*，则保留除相对寄存器符号外的所有局部符号。

用法

缺省情况下，汇编程序在其输出目标文件中描述的符号仅有:

- 导出的符号
- 重定位所依据的符号。

使用 KEEP 可保留有助于调试的局部符号。所保留的符号出现在 ARM 调试器和链接器映射文件中。

KEEP 不能保留寄存器相对的符号（请参阅第7-19 页的*MAP*）。

示例

```
label   ADC     r2,r3,r4
        KEEP    label         ; makes label available to debuggers
        ADD     r2,r2,r5
```

7.8.12 NOFP

NOFP 指令可确保在汇编语言源文件中没有浮点指令。

语法

```
NOFP
```

用法

使用 NOFP 可确保在软件或目标硬件不支持浮点指令的情况下，不使用任何浮点指令。

如果在 NOFP 指令后出现一个浮点指令，则会产生一个**未知的操作码**错误，并且汇编失败。

如果 NOFP 指令出现在浮点指令之后，则汇编程序产生错误：

Too late to ban floating point instructions(太迟了 ,无法禁止浮点指令)

并且汇编失败。

7.8.13 REQUIRE

REQUIRE 指令指定各节之间的相关性。

语法

REQUIRE *label*

其中：

label 是所需标签的名称。

用法

使用 REQUIRE 可确保包含了相关节（即使其不是直接调用的）。如果链接中包含了含有 REQUIRE 指令的节，则链接器也将包含含有指定标签定义的节。

7.8.14 REQUIRE8 和 PRESERVE8

REQUIRE8 指令指定当前文件要求堆栈八字节对齐。它设置 REQ8 编译属性以通知链接器。

PRESERVE8 指令指定当前文件保持堆栈八字节对齐。它设置 PRES8 编译属性以通知链接器。

链接器检查要求堆栈八字节对齐的任何代码是否仅由保持堆栈八字节对齐的代码直接或间接地调用。

语法

REQUIRE8 {*bool*}

PRESERVE8 {*bool*}

其中:

bool 是一个可选布尔常数，取值为 {TRUE} 或 {FALSE}。

用法

如果您的代码保持堆栈八字节对齐，在需要时，可使用 PRESERVE8 设置文件的 PRES8 编译属性。如果您的代码不保持堆栈八字节对齐，则可使用 PRESERVE8 {FALSE} 确保不设置 PRES8 编译属性。

注意

如果您省略 PRESERVE8 和 PRESERVE8 {FALSE}，汇编程序会检查修改 sp 的指令，以决定是否设置 PRES8 编译属性。ARM 建议明确指定 PRESERVE8。

您可以通过以下形式启用警告:

```
armasm --diag_warning 1546
```

有关详细信息，请参阅第3-2 页的*命令语法*。

您将会收到类似以下警告:

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
                                   breaks 8 byte stack alignment
       37 00000044         STMFD    sp!,{r2,r3,lr}
```


示例

```
REQUIRE8
REQUIRE8     {TRUE}      ; equivalent to REQUIRE8
REQUIRE8     {FALSE}     ; equivalent to absence of REQUIRE8
PRESERVE8    {TRUE}      ; equivalent to PRESERVE8
PRESERVE8    {FALSE}     ; NOT exactly equivalent to absence of PRESERVE8
```

7.8.15 ROUT

ROUT 指令标记局部标签的作用域边界（请参阅第3-26 页的*局部标签*）。

语法

{*name*} ROUT

其中:

name 是要分配给作用域的名称。

用法

使用 ROUT 指令可限制局部标签的作用域。这样就更容易避免意外引用错误的标签。如果区域中没有 ROUT 指令，则局部标签的作用域是整个区域（请参阅第7-65 页的*AREA*）。

使用 *name* 选项可确保每个引用都指向正确的局部标签。如果标签的名称或对标签的引用与前面的 ROUT 指令不匹配，则汇编程序会产生一条错误消息，并且汇编失败。

示例

```
            ; code
routineA    ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```